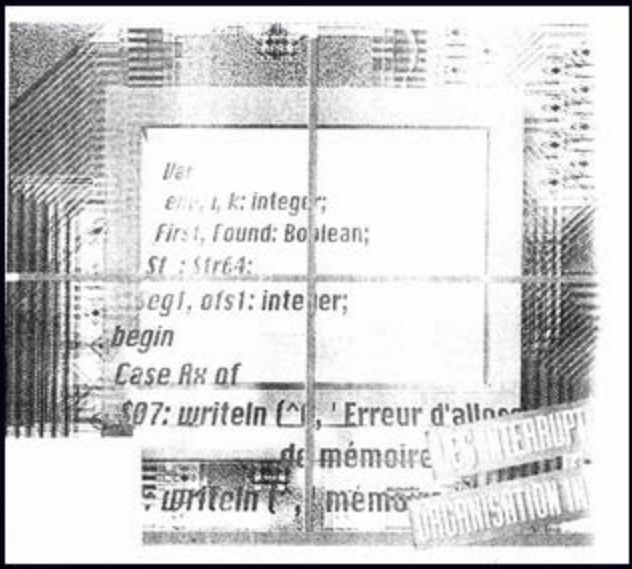
LES BEST-SELLERS
DE L'INFORMATIQUE

La maîtrise du MS-DOS® et du BIOS®



Ph. Mercier

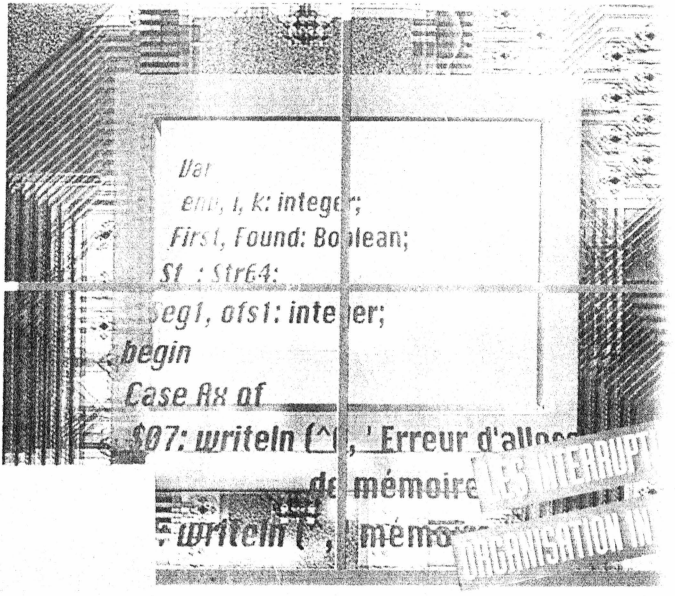
MARABOUT

LES BEST-SELLER DE L'INFORMATIQU

LA MAITRISE DU

MS-DOS®

ET DU BIOS®



Philippe Mercier

MARABOU
INFORMATIQ

CAMPAGNE INFORMATIQUE ANTI-VIRUS

Les virus sont partout et sont responsables de nombreuses maladies et dysfonctionnements des systèmes informatiques (bombes, problèmes d'impression, non reconnaissance du disque dur, plantages réguliers, etc.) :

Protégez-vous !
Ne les disséminez pas !

Comment faire ?

C'est très simple.
Les virus se propagent uniquement par **contact direct** (de disquette à disquette, de disquette à disque dur, de disque dur à disquette, de disque dur à disque dur) ou par **inoculation** (à travers les lignes téléphoniques).

Sachant cela, évitez tout moyen de contamination :

- ☛ verrouillez vos disquettes originales;
- ☛ verrouillez toute disquette contenant des fichiers importants ne pouvant être contaminés;
- ☛ ne chargez jamais sur le disque dur des programmes de source inconnue (et, de toute façon, vérifiez si la disquette n'est pas infectée);
- ☛ vérifiez régulièrement l'état de votre disque dur et de vos disquettes (il existe pour ce faire de nombreux programmes dans le domaine public);
- ☛ si c'est possible, installez dans le système de l'ordinateur un détecteur de virus (il en existe dans le domaine public);
- ☛ si votre disque dur ou vos disquettes sont contaminés, prenez les mesures qui s'imposent (l'idéal étant de reformater complètement le disque et de réinstaller les programmes).

Philippe Mercier

La maîtrise du MS-DOS® et du BIOS®

Les interruptions

Organisation interne

MARABOUT

Les collections *informatiques* sont dirigées par
Ghéorghiï Vladimirovitch Grigorieff

Direction de l'édition	Yann Delalande
Direction littéraire	Françoise Dequenne
Mise en page	Philippe Mercier
Composition	Les ateliers Virga Processing
Couverture	Duc Ho Minh

Composition et mise en page sur Word 3 ® à l'aide de la chaîne de micro-édition Apple (Mac II ® et Laserwriter NTX ®).

Word 3 est une marque déposée de Microsoft. Macintosh et LaserWriter sont des marques déposées d'Apple.

Tous les produits cités dans cet ouvrage sont des marques déposées ou des marques commerciales. L'auteur et l'éditeur déclinent toutes responsabilités pouvant provenir de l'usage des programmes figurant dans cet ouvrage.

© 1989 Marabout, Alleur (Belgique).

Toute reproduction d'un extrait quelconque de ce livre par quelque procédé que ce soit, et notamment par photocopie ou microfilm, est interdite sans l'autorisation de l'éditeur.

Sommaire

Préface	7
Chapitre 1 : Le système d'exploitation	11
Chapitre 2 : Les interruptions	19
Chapitre 3 : Les périphériques	49
Chapitre 4 : Les répertoires	105
Chapitre 5 : Manipulation de fichiers	135
Chapitre 6 : Organisation des fichiers sur disque	185
Chapitre 7 : La gestion de la mémoire et les programmes	241
Annexe 1 : Les codes d'erreur du MS-DOS	283
Annexe 2 : Liste par type des fonctions du MS-DOS	291
Annexe 3 : Glossaire	299
Index	307
Table des Matières	313

Préface

Dans un IBM PC, le système d'exploitation ne se contente pas seulement de la gestion des commandes simples comme DIR, COPY, etc.; il est également présent pour gérer efficacement l'ensemble des ressources du système. Ainsi, MS-DOS propose toute une série de "services", aussi appelés "interruptions", auxquels les programmes peuvent faire appel. Malheureusement, si la théorie apparaît relativement simple, il n'en est pas toujours ainsi de la pratique, et tout programmeur s'étant déjà aventuré dans ce domaine ne me contredira pas. Le but de cet ouvrage n'est donc pas de fournir une liste complète des interruptions mais plutôt d'essayer de comprendre, par la pratique, les méthodes de programmation utilisées pour gérer ces "services" (ou interruptions). Nous allons analyser comment MS-DOS organise les informations sur disque, comment il reconnaît et gère les différents types de périphériques, sa gestion de la mémoire, etc. Cela nous permettra de découvrir "sur le terrain" les principales interruptions et leur mode d'intervention dans les différentes situations rencontrées.

Cette analyse dévoilera également quelques-uns des secrets les plus intimes du PC ainsi qu'une série d'informations très intéressantes mais non décrites dans la documentation d'IBM ou de Microsoft. Notre objectif est de démontrer qu'une bonne connaissance

des interruptions permet de tirer un très grand profit des capacités de la machine et permet même de "faire sauter" certaines de ses limites! Cependant, il est évident que pour arriver à certains résultats — dignes des meilleurs programmeurs — il faudra vous armer de beaucoup de patience... La meilleure documentation ne vaudra jamais les heures de pratique et les nuits blanches passées à pianoter sur le clavier de l'ordinateur.

Nous espérons qu'après avoir lu cet ouvrage, vous comprendrez mieux comment fonctionnent les programmes et utilitaires les plus répandus (NORTON UTILITIES, PCTOOLS, FDISK, CHKDSK, etc.) Cette maîtrise de la machine vous permettra de réaliser de nombreux programmes qui vous paraissaient si compliqués peu de temps avant la lecture de ce livre.

Les connaissances requises

Tout le monde comprendra aisément que plus on va loin dans la découverte d'une machine, plus la programmation s'approche du hardware installé. Nous serons fréquemment amenés à faire appel à l'assembleur pour réaliser certains traitements particuliers. Une connaissance de ce langage ainsi que des structures de base d'un système informatique s'avère donc indispensable pour la lecture de cet ouvrage. De plus, nous avons voulu illustrer autant que possible la théorie par des exemples de

programmes écrits en TURBO PASCAL, ceux-ci pourront facilement être réécrits dans d'autres langages.
Les notions essentielles à connaître sont donc :

- L'assembleur

  Cela inclut les instructions de base du langage, les techniques d'adressage mémoire, les notations hexadécimale et binaire, la gestion de la pile et l'appel aux sous-programmes.

- Le TURBO PASCAL

  Cela inclut les instructions de base du langage et l'instruction INLINE.

Nous invitons le lecteur pour lequel ces notions présentent encore des difficultés à se reporter à d'autres ouvrages, du même auteur, parus dans cette collection (*L'Assembleur facile*, Marabout Service n° 885 et *Turbo Pascal facile*, Marabout Service n° 756).

Néanmoins, si vous êtes simplement curieux et souhaitez comprendre le mode de fonctionnement des interruptions, rien ne vous empêche de lire cet ouvrage en sautant les exemples de programmation.

Enfin, les lecteurs souhaitant disposer immédiatement des programmes décrits dans cet ouvrage peuvent se procurer la disquette comprenant la source des différents programmes analysés dans ce volume.

Chapitre 1

Le système d'exploitation

1. Introduction

En plus de la mémoire et du microprocesseur, l'ordinateur est équipé d'un clavier, d'un écran, d'un disque, etc. Le rôle du système d'exploitation MS-DOS est d'aider les programmes à communiquer avec ces différents éléments. Pour réaliser certains traitements particuliers, il faudra donc en connaître le fonctionnement général. Afin d'éviter que les programmes soient dépendants du matériel installé, les concepteurs du DOS ont pensé à découper le système d'exploitation en plusieurs parties :

- Les routines dépendantes du matériel sont regroupées dans le **BIOS**; elles peuvent varier d'un ordinateur à l'autre, ou en fonction de l'équipement.

- Les routines situées entre les programmes et le BIOS constituent **le noyau du système d'exploitation**; elles sont regroupées sous le nom générique "DOS" (Disk Operating System). Ainsi, les programmes n'accéderont plus directement à l'écran ou à l'imprimante, mais communiqueront avec le noyau du système d'exploitation en lui donnant instruction d'accéder à l'écran.

C'est ce concept qui est à l'origine de la compatibilité des ordinateurs IBM PC : un même programme pourra être exécuté sur des machines différentes pour autant qu'elles utilisent le même noyau du système d'exploitation.

2. Le Bios

BIOS est la dénomination abrégée de "Basic Input Output System" (système de base des entrées et sorties), c'est un ensemble de programmes de gestion de l'écran, du clavier, du disque, de l'imprimante, etc. **Ces programmes sont dépendants du matériel installé.** Et si on souhaite changer d'écran? Eh bien, le constructeur fournira avec l'écran un programme qui devra remplacer la routine de gestion de l'écran du BIOS. On pourra indiquer à l'ordinateur d'utiliser ce nouveau programme appelé "device driver" (contrôleur de périphérique) en plaçant la directive "DEVICE=..." dans un fichier appelé CONFIG.SYS. Les programmes comme "disk-manager" (pour utiliser un disque de plus de 32MB) ou les disques virtuels sont des exemples de drivers.

Note : Le système d'interruption offre l'avantage de ne pas devoir modifier les programmes en cas de modification de l'installation hardware: il suffira de changer les routines d'interruptions.

3. Le Ms-Dos

Le noyau du système d'exploitation est un ensemble de programmes beaucoup plus généraux que le BIOS, il est chargé de gérer au mieux les ressources de l'ordinateur (mémoire, périphériques); nous verrons qu'il met à la disposition des programmeurs un certain nombre d'outils fort précieux. Les interruptions utilisées par MS-DOS sont comprises entre 20h et 3Fh. On peut distinguer 5 catégories de services :

1. La gestion des programmes.
2. La gestion de la mémoire.
3. La gestion des périphériques (avec l'aide du BIOS).
4. La gestion des fichiers.
5. L'Interface utilisateur.

La gestion des programmes

Cela inclut le chargement des programmes en mémoire (appel aux fonctions DISQUE du BIOS), le lancement de l'exécution et la terminaison. Bien que MS-DOS ne soit pas un système multitâche, nous verrons qu'il a la possibilité de gérer plusieurs programmes en mémoire à un moment déterminé et qu'il pourra passer de l'un à l'autre sous certaines conditions. Pour mieux comprendre comment tout cela fonctionne, nous serons amenés à analyser les techniques d'allocation de mémoire et la communication entre

programmes (cf. le chapitre 7 consacré à la mémoire).

La gestion de la mémoire

Il existe toute une série de fonctions permettant à un programme de demander de la mémoire; la mémoire pourra même être attribuée à plusieurs programmes en même temps. La gestion de la mémoire est directement liée à la gestion des programmes (cf. le chapitre 7).

La gestion des périphériques

Le DOS propose un certain nombre de fonctions permettant d'optimiser la gestion des périphériques (cf. le chapitre 3).

La gestion des fichiers

Un des rôles principaux du MS-DOS, et des systèmes d'exploitation en général, est de s'occuper de l'organisation des fichiers sur disque : nous analyserons plus tard la structure d'un disque et les méthodes de stockage des fichiers (cf chapitres 4, 5 et 6).

L'interface utilisateur

Finalement, il existe un programme appelé COMMAND.COM, c'est le "processeur de commandes", il

fait partie intégrale du système d'exploitation. On distingue 2 types de commandes :

- Les commandes **internes** (dir, copy, etc.) traitées directement par ce programme.

- Les commandes **externes** qui sont des programmes externes (format, diskcopy, etc.).

Il faut aussi savoir que COMMAND.COM est le programme général de contrôle de l'ordinateur. Il est chargé en mémoire au démarrage et ne se terminera jamais : quand un autre programme est exécuté, COMMAND.COM lui donne le contrôle de l'ordinateur puis le récupère aussitôt ce programme terminé (cf. chapitre 7 : La mémoire; exécution d'un "programme fils").

Chapitre 2

Les interruptions

Les différents services (on parle aussi de routines, de sous-programmes) proposés par le système d'exploitation (BIOS et DOS) sont identifiés par un **numéro**. Il s'agit de sous-programmes qui seront exécutés à la demande du microprocesseur, d'un programme ou d'un périphérique. Dans le cas des demandes émanant du processeur ou d'un périphérique, le programme en cours sera ***interrompu*** pour exécuter la routine souhaitée; c'est pour cette raison que ces routines sont aussi appelées **interruptions**. Certaines interruptions disposent de plusieurs services, on parlera alors de **fonctions.**

- Les services du BIOS et du DOS sont appelés **interruptions.**

- Les services d'une interruption sont appelés **fonctions.**

Les interruptions "processeur"

Ces interruptions (routines) sont exécutées par décision du microprocesseur.

Int	Désignation
00h	Appelée en cas de division par 0 (*).
01h	Exécution pas à pas (utilisée par des programmes comme DEBUG) (*).
02h	Appelée en cas d'erreur mémoire (parité = chip mémoire défectueux), provoque un arrêt du système (*).
03h	Point d'arrêt (utilisé par des programmes comme DEBUG pour permettre d'interrompre l'exécution d'un programme à une instruction déterminée) (*).
04h	Appelée en cas de dépassement de capacité (*).

(*) Interruptions BIOS

Nous verrons qu'il est possible de "masquer" temporairement les interruptions, ce qui aura seulement pour effet de les différer car, comme cela apparaîtra plus tard, certains traitements critiques ne pourront en aucun cas être suspendus. L'interruption 02h appartient à une catégorie d'interruptions, appelées "*Non masquable interrupts*", qui peuvent demander une attention immédiate du processeur: il est IMPOSSIBLE de les masquer. Ce type de demande est en général très grave et provoque un arrêt instantané de la machine (chip défectueux, erreur d'alimentation, etc.).

Les interruptions "périphériques"

Toute une série de tâches doivent être exécutées pour assurer le bon fonctionnement de l'ordinateur :

- **Il faut exécuter les instructions du programme en cours.**
- **Il faut lire les caractères tapés au clavier.**
- **Il faut gérer les demandes d'accès disque.**
- **Il faut maintenir l'heure exacte.**
- Etc.

Pour que la machine soit rapide, on évitera toute action inutile. Ainsi, il ne servira à rien d'aller vérifier toutes les 10 millisecondes s'il y a un caractère tapé au clavier; par contre, si on ne va pas voir au bon moment, le caractère risque d'être écrasé par le suivant. Pour résoudre ce problème, le processeur est entouré de "collaborateurs" dont le seul rôle est de lui signaler qu'un périphérique requiert son attention. Dès qu'il reçoit cet avertissement, il **interrompra** l'exécution du programme en cours pour s'occuper du périphérique, plus précisément, pour exécuter **l'interruption associée au périphérique**. Grâce à ce système, la routine du BIOS sera exécutée *juste quand il faut*, il n'y a donc plus de perte de temps. Les interruptions qui traitent les demandes émanant des périphériques sont appelées "**hardware**" car elle sont liées directement soit au processeur soit à la carte mère de l'ordinateur. Elles sont contrôlées par un chip appelé 8259 (contrôleur d'interruption) ou PIC (programmable interrupt controler). Ce *chip* peut gé-

rer jusqu'à 8 dispositifs provoquant des interruptions, chaque dispositif possède une entrée distincte appelée "niveau d'interruption" :

Niveau	Int	Désignation
0	08h	Timer (horloge) (*)
1	09h	Clavier (*)
2	0Ah	Réservé
3	0Bh	Com2 (port série 2) (*)
4	0Ch	Com1 (port série 1) (*)
5	0Dh	Disque dur (*)
6	0Eh	Disque souple (*)
7	0Fh	Imprimante (*)

(*) Interruptions BIOS

Le contrôleur attribue une **priorité** à chacune des interruptions: l'interruption de niveau 0 possède la plus grande priorité. Ce système permet au 8259 de gérer le cas de deux interruptions (demande d'un périphérique) qui arriveraient en même temps. De plus, si le 8088 est en train de traiter une interruption, le 8259 empêche un périphérique ayant une priorité inférieure d'interrompre son travail. Il est aussi possible qu'un programme empêche les interruptions (c'est nécessaire lors de certaines opérations dangereuses); le bit 9 du registre FLAG indique si les interruptions sont ou ne sont pas autorisées, c'est le principe du masquage que nous avons déjà énoncé plus haut.

Les interruptions "programmes"

Ces interruptions sont invoquées par les programmes qui souhaitent profiter des services du BIOS ou du DOS, on les appelle "interruptions logicielles".

Int	Désignation
05h	Impression de l'écran (*)
10h	Gestion du vidéo (*)
11h	Détermination de l'équipement installé (*)
12h	Détermination de la mémoire installée (*)
13h	Gestion des secteurs des disques (*)
14h	Gestion des ports série (*)
15h	Gestion des lecteurs de cassettes (*)
16h	Gestion du clavier (*)
17h	Gestion de l'imprimante (*)
18h	Basic (*)
19h	Redémarrage de l'ordinateur (boot) (*)
1Ah	Gestion de l'heure (*)
1Bh	Gestion du Ctrl-Break (*)
1Ch	Compteur d'horloge (*)
1Dh	Paramètres du vidéo (*)
1Eh	Paramètres disquette (*)
1Fh	Table de caractères graphiques (*)
20h	Terminaison de programme
21h	Appel à MS-DOS
22h	Point de terminaison d'un programme
23h	Gestion du CTRL-C
24h	Gestion des erreurs critiques
25h	Lecture logique de secteurs sur disque
26h	Ecriture logique de secteurs sur disque
27h	Installation d'un programme *résident*
28h	MS-DOS en attente
2Fh	Spooler d'imprimante (Print.com)
40h	Contrôleur de disquettes (*)
41h	Paramètres disque dur (*)

(*) Interruptions BIOS

Nous verrons que les adresses des routines d'interruption sont stockées au début de la mémoire. Dans les emplacements prévus pour les interruptions **1Dh, 1Eh, 1Fh**, on trouvera, en fait, les adresses de tables utilisées par les programmes de la ROM qui contiennent les informations nécessaires au contrôle de certains périphériques. On notera par exemple que la commande MODE qui permet de modifier le nombre de colonnes de l'écran (40 ou 80) modifie les valeurs de la table "vidéo"; dans la table "disquette", on trouvera une valeur indiquant le temps pendant lequel le moteur de la disquette doit continuer à tourner après une opération, etc.

Note: A l'inverse des interruptions "hardware" qui arrivent de manière tout à fait imprévisible, **les interruptions software sont contrôlées par programme.**

Le vecteur d'interruptions

Comme nous l'avons vu, le périphérique ou le programme qui fait la demande d'interruption aura juste besoin de spécifier un numéro. Le processeur pourra trouver l'adresse de la routine à exécuter grâce à une table située tout au début de la mémoire, cette table est appelée "table des vecteurs d'interruptions". Elle a une dimension de 1024 octets et peut donc stocker les adresses de 256 routines (une adresse = segment:offset qui sont deux quantités de 16 bits). Un élément de la table qui contient l'adresse d'une routine est appelé "vecteur d'interruption". Pour calculer l'offset (le déplacement) de l'adresse d'une interruption dans la table, il suffit de multiplier par 4 (car il y a 4 bytes par adresse) le numéro de l'interruption:

```
Interruption 0 ==> Offset 0000h=(0*4)
Interruption 1 ==> Offset 0004h=(1*4)
Interruption 2 ==> Offset 0008h=(2*4)
Interruption 3 ==> Offset 000Bh-(3*4)
...
```

Lorsque le microprocesseur devra exécuter une des routines, il pourra consulter l'adresse du programme dans le vecteur d'interruption approprié. Cette méthode permettra aussi, comme nous le verrons plus tard, de modifier les interruptions : en fait on installera dans le vecteur d'interruption (à l'endroit souhaité) l'adresse d'une routine personnelle.
Pourquoi ? Surprise...

Appel d'une interruption

Méthodes d'appel

L'appel à une interruption sera géré à peu près comme l'appel à un sous-programme :

- Pour appeler une interruption à partir d'un programme en assembleur, il faudra utiliser l'instruction INT (équivalent d'un CALL).

- Pour appeler une interruption à partir d'un langage de haut niveau comme TURBO PASCAL, on pourra utiliser une procédure interne: INTR, et on définira les registres comme suit :

```
TYPE
 Registres = record
    {définition des registres du micro-  }
    {processeur                          }
  Case Integer of
  1:(ax,bx,cx,dx,bp,si,di,ds,es,flags:Integer);
  2:(al,ah,bl,bh,cl,ch,dl,dh :byte);
  end;

VAR
 Reg:Registres;
```

Transmission de paramètres

Pour la plupart des interruptions, on devra spécifier des paramètres d'entrée; le résultat, lui, sera renvoyé par des paramètres de sortie : ces paramètres sont transmis par les registres.

Exemples :

- Appel de l'interruption **10h**, fonction **00h** :

```
MOV   AH,00h  ; fonction 00h
MOV   AL,6    ; paramètre
INT   10h     ; Appel de l'interruption
```

- Appel de l'interruption **16h**, fonction **02h** :

```
MOV   AH,02h  ; Fonction 02h
INT   16h     ; Appel de l'interruption
```

- Appel de l'interruption **17h**, fonction **02h** :

```
MOV   AH,02h  ; Fonction 02h
MOV   DX,00h  ; Paramètre
INT   17h     ; Appel de l'interruption
```

Etat de la pile lors de l'appel

En cas d'appel à une interruption, le microprocesseur exécutera toujours l'équivalent des instructions suivantes :

```
PUSHF           ;Met le registre FLAG sur
                ;le stack
CLI             ;Empêche les interruptions
CALL adresse    ;Transfert à la routine,
                ;adresse lue dans le
                ;vecteur d'interruptions.
```

Le 8088 préserve donc les indicateurs ainsi que l'adresse de la prochaine instruction de programme à exécuter. **Tous les registres utilisés par l'interruption devront être sauvegardés et restaurés par celle-ci.** Les registres qui contiendront les résultats seront bien sûr détruits; il appartient donc au programme qui appelle l'interruption de les conserver sur la pile avant l'appel. L'état de la pile au moment où l'interruption reçoit le contrôle sera donc :

```
        Flag    Contenu de FLAG
        Cs      Valeur de CS au moment de
                l'appel
SP-->   Ip      Valeur de IP au moment de
                l'appel
```

Modification d'une interruption

Il n'est bien entendu pas possible de modifier les routines du BIOS puisqu'elles se situent en ROM (mémoire à lecture uniquement). Par contre, on pourra modifier les vecteurs d'interruptions et installer l'adresse d'une nouvelle routine qui sera alors exécutée à la place de la routine de la ROM.

- La fonction **25h** de l'interruption **33h** permet de modifier l'adresse d'une interruption.

- La fonction **35h** de l'interruption **33h** permet de lire l'adresse d'une interruption.

Lorsqu'un programme souhaite installer sa propre routine d'interruption, il devra suivre la procédure suivante :

1 - Appeler la fonction **35h** de l'interruption **33h** pour lire l'adresse de la routine, cette adresse devra être sauvegardée.

2 - Appeler la fonction **25h** de l'interruption **33h** pour installer l'adresse de la nouvelle routine.

3 - Avant de se terminer, le programme devra restaurer l'adresse originale (avec la fonction 25h).

On pourra aussi créer des programmes *résidents* qui modifient les interruptions.

La fonction **31h** de l'interruption **21h** installe un programme en mode "*résident*" (cf. exemple ci-après, page 34).

Note : Les points 1 et 3 sont inutiles si on installe le programme en mode "*résident*".

Méthodes de fin d'exécution d'une interruption

Il existe plusieurs possibilités de "retour" pour une routine d'interruption et il est primordial de les connaître :

- La dernière instruction de la routine est l'instruction **IRET** (retour d'interruption), son effet sera le même que l'instruction RET à l'exception près que le registre FLAG sera automatiquement retiré de la pile (équivalent de POPF) et donc rétabli à sa valeur originale.

- La routine utilise le registre FLAG pour renvoyer une information (c'est le cas notamment de l'interruption 13h), dans ce cas le registre FLAG qui a été mis sur la pile au moment de l'appel à l'interruption est éjecté par une instruction de fin: **RET 2.** Le programme appelant qui souhaite conserver le contenu de FLAG devra donc le mettre sur le stack avant d'appeler les routines qui détruisent le FLAG (celles qui renvoient des informations dans le FLAG).

- La routine utilise le FLAG pour renvoyer des informations mais **elle ne l'éjecte pas de la pile !!!** C'est le cas des interruptions 25h et 26h, il faudra donc que le programme appelant exécute une instruction POP ou ADD SP,2 pour rétablir la stabilité de la pile... Si ce n'est pas fait, un crash est assuré très rapidement !!

Exemple de programmation

A titre d'exemple, nous allons modifier l'interruption 21h (appel à MS-DOS) pour qu'elle affiche "21" en haut à droite de l'écran quand le processeur exécute l'interruption 21h. Pour cela, nous allons installer l'adresse de notre routine dans le vecteur d'interruption approprié et rendre notre routine "résidante". Dès que l'installation sera terminée, le contrôle sera rendu à COMMAND.COM qui réaffichera le "prompt" du DOS et se mettra en attente de la prochaine commande.

A ce stade, tout programme qui fera appel à l'interruption **21h** appellera *notre* routine!

Pour ce mode d'attente, COMMAND.COM utilise une fonction de l'interruption 21h, ce qui justifie que "21" reste affiché en permanence à l'écran. Pour constater que le système marche bien, on pourra par exemple exécuter un traitement de texte quelconque; le "21" devrait disparaître quand on tape du texte et apparaître un bref instant au moment où on sauvegarde le texte, quand on demande la liste des fichiers, etc.

De plus, pour éviter une double installation, le programme vérifiera s'il n'est pas déjà installé. Pour cela, nous avons ajouté une fonction à l'interruption **21h** : la fonction **FFh**. Elle renverra Ah=0 si la routine est installée et ne fera rien dans le cas contraire puisque cette fonction n'existe pas dans l'interruption **21h** d'origine !

Note: L'interruption 21h renvoie des résultats par l'intermédiaire du registre FLAG, la méthode de retour à utiliser est donc un RET 2.

```
Code      Segment
          Org  100h
          Assume cs:code
Start:    Jmp  Installe

CR        Equ  0Dh
LF        EQU  0Ah

; liste des messages :

Welcome   DB   CR,LF
          DB   'Int 21 Monitor Program'
          DB   CR,LF
          DB   'Copyright (C) 1989  Ph. Mercier'
          DB   CR,LF,CR,LF,'$'

DejaInst  DB   'Programme déjà installé.',CR,LF,'$'
InstOk    DB   'Installation terminée.',CR,LF,'$'

; Zones de sauvegarde :

Int21     Label DWORD      ;Sauve l'adresse de
                           ;l'ancienne routine
Int21Off  DB   0
INT21Seg  DB   0

Save1     DB   0
Save2     DB   0

; Notre Interruption 21h, appel à MS-DOS:
; ---------------------------------------

TsrInt21  PROC FAR
          Sti

; Regarde si c'est notre fonction FFh:
          Pushf
          Cmp  Ah,0
          Jne  DosFcts     ; non, continue
```

```
            Popf
            Mov  Ah,0          ; oui, renvoie 0 dans AH
            Ret  2             ; retour

DosFcts:    Popf
            Push Es
            Push Ax
            Mov  Ax,0B000h     ;adresse écran
                               ;monochrome
            Mov  Es,Ax

  ;sauve les caractères actuellement en colonnes
  ;79 et 80:
            Mov  al,ES:156     ;colonne 79
            Mov  Save1,Al      ;sauve caractère actuel
            Mov  Al,Es:158     ;colonne 80
            Mov  Save2,AL      ;sauve caractère actuel
 ; Ecrit '21':
            Mov  Byte Ptr Es:156,'2'
            Mov  Byte Ptr Es:158,'1'
            Pop  Ax
            Pop  Es


; Appelle l'interruption 21h d'origine:
            Pushf
            Cli
            Call Int21

 ; Restaure les caractères en colonnes 79 et 80:

            Push Es
            Push Ax
            Mov  Ax,0B000h  ;adresse écran
                            ;monochrome
            Mov  Es,Ax
            Mov  Al,Save1
            Mov  Es:156,Al
            Mov  Al,Save2
            Mov  Es:158,Al
            Pop  Ax
            Pop  Es

            Ret  2          ;retour
```

```
TsrInt21 ENDP

EndOfResident:          ;Fin de la partie que l'on
                        ;laisse résidante en mémoire,
                        ;toute la procédure
                        ;d'installation est
                        ;supprimée.

; Procédure d'installation du programme résident:
; -----------------------------------------------

Install:

        Mov  Ax,cs
        Mov  Ds,ax

;Affiche message de bienvenue

        Mov  dx,offset welcome
        Mov  ah,09h
        Int  21h

; Teste si déjà installé:

        Mov  Ah,0ffh            ; Notre fonction
        Int  21h                ; appel
        Cmp  Ah,0               ; 0?
        Je   Twerr              ; oui,déjà installé

; Sauvegarde adresse de l'ancienne int21 et
; installe notre routine à la place:

        Mov  ax,3521h           ; lire l'adresse
        Int  21h                ; appel
        Mov  Int21Off,bx        ; sauvegarde
        Mov  Int21Seg,Es

        Mov  ax,2521h           ; Ecrire notre
                                ; adresse
        Mov  dx,offset TsrIntr21
        Int  21h

; Termine et installe ce programme en mode
; résident:

        Mov  dx,offset instOk
```

```
        Mov   ah,09h
        Int   21h

; Calcule la taille du programme à laisser
; résident:
        Mov   dx,EndOfResident-start+100h+15
        Mov   cl,4
        shr   dx,cl       ; divise par 4 pour
                          ; obtenir une quantité
                          ; exprimée en paragraphes.
        mov   al,0
        Mov   ah,31h      ; fonction "résidant"
        Int   21h         , appel
```

termine, une erreur s'est produite:

```
Twerr:  Mov   dx,offset dejainst
        Mov   ah,09h
        Int   21h

        Mov   ah,4Ch      ; renvoie un code
                          ; au parent (cf. chap 7,
                          ; la mémoire)
        Mov   al,0FFh
        Int   21h

Code    ends
        end start
```

Evolution du MS-DOS

Comme nous l'avons signalé, il y a une différence fondamentale entre le BIOS et le DOS proprement dit (le noyau). Le BIOS contient notamment des routines de gestion des périphériques. Le DOS est l'intermédiaire entre les programmes et le BIOS; on pourra y faire appel au moyen de l'interruption 21h qui dispose d'un nombre impressionnant de fonctions.

Il faut cependant remarquer que les programmes pourraient très bien faire directement appel aux interruptions du BIOS mais, pour des raisons de compatibilité, ce n'est pas conseillé. Nous consacrerons donc la majeure partie de cet ouvrage à étudier les fonctions de l'interruption 21h, et dans certains cas plus précis, nous nous intéresserons directement aux interruptions du BIOS.

Si nous nous référons à la liste des fonctions de l'interruption **21h** et que nous l'analysons en détail, il y a de quoi décourager les plus coriaces : beaucoup de fonctions semblent redondantes et on a du mal à trouver une continuité, cette série de fonctions ressemble à tout, sauf à un ensemble de services bien organisé ! Comment est-il possible qu'un système vendu à des millions d'exemplaires puisse être défini de la sorte ? La réponse, nous allons la trouver en analysant l'histoire de ce système d'exploitation...

L'histoire de MS-DOS commence au temps où le système d'exploitation CPM règnait en "maître" quasi ab-

solu sur beaucoup de petits ordinateurs. A ce moment, les programmes les plus répandus étaient conçus pour ces machines-là. La société Microsoft ayant été chargée par IBM de concevoir un système d'exploitation pour ce qui allait devenir le PC que nous connaissons aujourd'hui a eu un raisonnement fort logique : si MS-DOS est semblable à CPM, les sociétés qui ont développé les programmes pour CPM pourront facilement les adapter pour MS-DOS et l'introduction du PC sur le marché en sera facilitée. Aussitôt dit, aussitôt fait, MS-DOS fait son apparition sur le marché et les concepteurs de programmes s'empressent de suivre : il y a IBM derrière, la machine va certainement avoir du succès...

Ce fut manifestement le cas et, au fur et à mesure que les besoins se créaient, que les disques devenaient de plus en plus rapides et qu'on avait toujours besoin de plus de capacité, il apparaissait évident que les techniques développées par CPM, et reprises sous MS-DOS, n'étaient pas parmi les meilleures; il fallait trouver autre chose...

Un des problèmes les plus importants concernait la méthode de gestion des fichiers du CPM. Pour chaque fichier utilisé par un programme, le système avait besoin d'une table appelée FCB (File Control Block) contenant des informations sur ce fichier : le disque, le nom du fichier, un pointeur indiquant la position courante dans le fichier... Les informations étaient transférées du disque vers une zone mémoire appelée DTA (Disk Transfert Area) dont l'adresse se trouvait dans une autre zone mémoire de 256 bytes, le PSP

(program segment prefix), précédant le programme en mémoire. Le problème principal des FCB est qu'il n'y avait pas de place pour stocker le chemin (path) indiquant dans quel répertoire se trouvait le fichier, et c'est à cause de cela que les versions 1.xx ne permettaient d'avoir qu'un seul répertoire par disque ! Les périphériques standard (clavier, imprimante, écran) étaient manipulés de manière tout à fait différente des fichiers sur disque alors qu'ils présentent les mêmes caractéristiques (on les appelle fichiers "logiques", ce qui prouve qu'il s'agit aussi de fichiers !).

A l'époque des versions 1.xx de MS-DOS, le système ne proposait que très peu de fonctions dans l'interruption 21h; on y trouvait des fonctions d'accès aux différents périphériques standard (au moins une fonction pour chaque périphérique), des fonctions de gestion des fichiers (création, effacement, sélection du disque courant, modification de l'adresse de la DTA,...), des fonctions de gestion de l'heure et de la date et des fonctions permettant de lire les adresses du vecteur d'interruptions.

Puis, un beau jour, on vit apparaître la version 2.0... et le monde s'émerveilla (?)... Microsoft avait complètement révisé sa technique de gestion des fichiers pour implanter une méthode, plus proche de systèmes comme UNIX. Cette méthode présentait un nombre considérable d'avantages par raport à la technique précédente. On ne parlera désormais plus de FCB mais de "***handle***". Lorsqu'un programme souhaitera travailler avec un fichier, MS-DOS lui attribuera un numéro (le handle), c'est ce *handle* que le programme

devra spécifier avant chaque opération concernant le fichier. Ce système supporte sans restriction les chemins et il ne faut pas obligatoirement se trouver positionné dans le répertoire courant pour pouvoir accéder au fichier. De plus, les périphériques standard sont gérés de la même manière. La seule différence est qu'un numéro leur est attribué d'office et qu'il ne faudra pas demander à MS-DOS "d'ouvrir" un de ces fichiers avant l'utilisation (contrairement aux fichiers résidant sur disque).

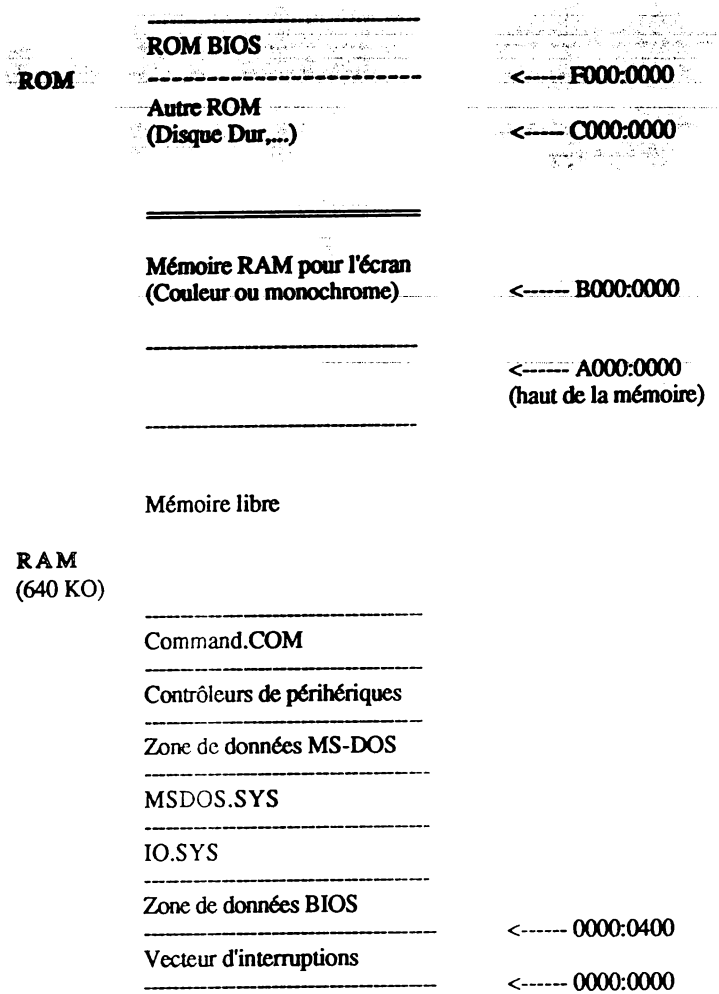
Dans cette nouvelle version de MS-DOS, on vit l'apparition des répertoires permettant de classifier les fichiers par types et, dès lors, l'interruption **21h** disposa d'un nombre beaucoup plus important de fonctions: en plus des fonctions disponibles en version 1.xx, on pouvait trouver des fonctions de gestion des répertoires (création, effacement, changement de répertoire), des fonctions sur les handles, bien entendu (création, effacement, lecture, écriture, etc.), des fonctions permettant de lire ou de modifier les attributs d'un fichier, une fonction très complète de dialogue avec le programme qui contrôle un périphérique et des fonctions très intéressantes de gestion de la mémoire et des programmes.

Finalement les versions **3.xx** de MS-DOS proposèrent en plus quelques fonctions de gestion de la mémoire et des programmes, une fonction permettant de mieux diagnostiquer les cas d'erreur et surtout des possibilités de travailler en réseau local.

On peut maintenant comprendre pourquoi nous parlions de "désordre" au sein des fonctions de l'interruption 21h. Le système a tellement évolué (en bien) depuis la version 1.00 et, étant donné qu'il a été nécessaire que MS-DOS reste compatible avec lui-même (!), Microsoft a conservé les fonctionnalités originales. Il y a bien entendu certaines fonctions de la version 1 qui restent intéressantes mais toutes les fonctions utilisant les FCB et les fonctions de gestion des périphériques standard peuvent être tout simplement oubliées. C'est ce que nous allons faire au cours de cet ouvrage car nous estimons qu'il y a suffisamment à dire sur les fonctionnalités actuelles pour ne pas s'étendre sur "comment ça marchait avant..." Nous verrons cependant que dans certains cas, il faudra faire une petite exception, et pour ceux que cela intéresserait, les fonctions de la version 1.0 sont listées en fin de livre, parmi les autres, mais à titre indicatif uniquement.

Organisation de la mémoire

Avant de continuer, nous devons avoir en tête le schéma de la mémoire. Lorsque l'ordinateur est allumé (ou qu'on exécute le classique CTRL-ALT-DEL), le contrôle est transmis à un programme d'amorçage ("bootstrap") dont la première instruction se trouve **toujours** à l'adresse FFFF:0000 (c'est une caractéristique du microprocesseur 8088 et pas du système MS-DOS). Cette adresse est en mémoire ROM. Ce programme en chargera un autre situé sur le premier *secteur physique* du disque (cf. chapitre sur les disques) et lui donnera le contrôle. Il y aura alors vérification que le disque contient bien le système dans deux fichiers IO.SYS et MSDOS.SYS (ou IBMIO et IBMDOS). Ces deux fichiers sont chargés en mémoire et le contrôle est donné à IO.SYS qui procédera à une phase d'initialisation (en consultant CONFIG.SYS notamment). Dès que cette phase est terminée, le programme COMMAND.COM est chargé en mémoire et exécuté, c'est lui qui se chargera d'exécuter AUTOEXEC.BAT et se mettra ensuite en attente d'une commande après avoir affiché le "prompt" du DOS. A ce stade, on a le schéma mémoire suivant :



En voyant ce schéma, on comprend pourquoi on parle de 640Ko bien que le 8088 puisse adresser 1MB, il s'agit de 640Ko de mémoire destinée à l'utilisateur (RAM), le reste étant réservé au système lui-même (ROM, mémoire écran...).

Récapitulation

- Les services du DOS et du BIOS sont appelés **Interruptions.**
- Les services au sein d'une interruption sont appelés **fonctions.**
- Les demandes d'interruptions peuvent émaner du processeur, d'un programme ou d'un périphérique.
- La table des vecteurs d'interruption est stockée tout au début de la mémoire. Chaque élément a une longueur de 4 bytes et contient l'adresse d'une interruption.
- Pour appeler une interruption, il faudra utiliser l'instruction assembleur **INT** ou, en Turbo Pascal, l'instruction **INTR.**
- Lors de l'appel à une interruption, les paramètres sont transmis par les registres.
- La fonction 25h permet de modifier l'adresse d'une interruption et la fonction 35h permet de la lire. Ces fonctions font partie de l'interruption **21h.**

☛ Il existe trois méthodes de fin d'exécution d'une interruption :

- Par un IRET.
- Par un RET 2, le registre flag est donc détruit au retour de cette interruption.
- Par un RET, il faudra alors rétablir la stabilité de la pile par une instruction ADD SP,2 juste après le retour de l'interruption.

Chapitre 3

Les périphériques

1. Introduction

De manière schématique, les ordinateurs sont composés d'une unité centrale (microprocesseur), de mémoire et, évidemment, de périphériques pour communiquer avec le monde extérieur. Dans un ordinateur IBM PC, on aura:

- Le clavier (périphérique d'entrée).
- L'écran (périphérique de sortie).
- L'imprimante (périphérique de sortie).
- Les unités de disquette ou disque dur (pour stocker les informations).

On pourra aussi connecter d'autres types de périphériques comme:

- Une souris.
- Un modem.
- Une unité de sauvegarde (tape streamer).
- Un joystick.
- Une table traçante.
- Etc.

Les périphériques peuvent être regroupés en deux catégories:

- Les périphériques orientés "**caractères**".
- Les périphériques orientés "**bloc de caractères**".

2. Les périphériques "caractères"

Ces périphériques sont prévus pour ne travailler qu'avec un seul caractère à la fois, ils sont identifiés par un nom. Parmi les périphériques "caractères", on trouvera :

- Le clavier (kbd ou con).
- L'imprimante (lpt1, lpt2, ... ou prn).
- L'écran (Con).
- La sortie série (modem) (Aux ou Com1, Com2).

Les périphériques font partie du matériel (hardware), il sont donc accessibles au moyen des interruptions (dialogue direct avec le BIOS) ou par l'intermédiaire d'une des fonctions du MS-DOS (fonction de l'interruption **21h**). Les interruptions du BIOS et les fonctions qu'elles proposent sont les suivantes :

L'interruption **10h** - Gestion de l'écran
L'interruption **14h** - Gestion du port série
L'interruption **16h** - Gestion du clavier
L'interruption **17h** - Gestion de l'imprimante

Comme nous le verrons dans un prochain chapitre, les périphériques peuvent être traités comme des **fichiers logiques**, chaque fichier utilisé par un programme recevra un numéro appelé **"handle"**. On

pourra dès lors utiliser les fonctions du MS-DOS de lecture (**3Fh**) et d'écriture (**40h**). Un des principaux avantages du système des handles est la possibilité de redirection des entrées/sorties.

- La fonction **3Fh** de l'interruption **21h** autorise la lecture sur un périphérique.

- La fonction **40h** de l'interruption **21h** autorise l'écriture sur un périphérique.

L'interruption 10h - gestion de l'écran

Comme chacun sait, il est possible d'installer plusieurs types d'écrans sur les ordinateurs IBM PC, les configurations les plus courantes étant :

- L'adaptateur monochrome.
- L'adaptateur CGA (couleur graphique).

➥ L'adaptateur monochrome

Ce mode vidéo autorise un affichage "noir et blanc" de 25 lignes et 80 colonnes. Chaque caractère affiché à l'écran est constitué de deux éléments de 1 octet :

- Le code ascii du caractère.
- Un attribut (gras, normal, souligné, inversé).

Les octets pairs contiennent les attributs et les impairs contiennent les codes ascii. L'adresse du début de la mémoire vidéo est B000h, le premier caractère sera donc stocké en B000h:0001h, le second en B000h:0003h, le premier caractère de la seconde ligne à l'adresse B000h:0161, etc.

➥ L'adaptateur graphique

Cet adaptateur possède trois modes :

- Un mode qui divise l'écran en 25 lignes de texte et 80 colonnes de texte comme l'adaptateur monochrome (mode texte).
- Un mode qui divise l'écran en 200 lignes et 320 colonnes (moyenne résolution).
- Un mode qui divise l'écran en 200 lignes et 640 colonnes (haute résolution).

Le mode texte est géré comme l'adaptateur monochrome, la seule différence est que l'adresse du début de la mémoire vidéo est B800h. En mode "haute résolution", on disposera de deux couleurs et de quatre en mode "moyenne résolution". L'interruption **10h** permet de gérer ces deux modes vidéo, elle dispose de 16 fonctions dont les plus importantes sont :

00h Initialisation du mode vidéo.
01h Définition du curseur.
02h Initialisation de la position du curseur.
03h Lecture de la position du curseur.
08h Lecture du caractère (et de son attribut) se trouvant à la même position que le curseur.
09h Ecriture d'un caractère et de son attribut à la position courante du curseur.
0Ah Ecriture d'un caractère à la position courante du curseur.
0Ch Ecriture d'un pixel.
0Dh Lecture d'un pixel.
0Fh Lecture du mode courant (texte, graphique, etc.).

Exemples de programmation

- **Les deux routines suivantes, écrites en Pascal, seront utilisées pour éteindre et allumer un curseur:**

```
Type
 RegType = Record
  Case Integer Of
   1:(AX,BX,CX,DX,BP,SI,DI,DS,ES: Integer);
   2:(AL,AH,BL,BH,CL,CH,DL,DH:Byte);
  end;

Procedure CursorOff(Var CT:Integer);
 {Sauve et éteint le curseur }
Var
 Curs:Integer Absolute $40:$60;
 Reg:Regtype;
Begin
  Ct:=Curs;          {Sauve le curseur actuel}
  Reg.ch:=$20;       {Code pour "pas de curseur"}
  Reg.ah:=1;         {Fonction 1, spécifie le
                     {format du curseur }
  Intr($10,reg);     {appelle l'interruption}
end;{CursorOff}


Procedure CursorOn(CT:Integer);
 { Rétabli un curseur préalablement sauvé }
 { par CursorOff }
Var
 Reg:Regtype;
Begin
 Reg.cx:=Ct;         {Ancien Curseur}
 Reg.ah:=1;          {Fonction 1, spécifie le
                     {format du curseur }
 Intr($10,reg);      {Appelle l'interruption}
end;{CursorOn}
```

Et l'appel sera:

```
Var
 SauveCurs:Integer;
begin

 CursorOff(SauveCurs);

... insérer ici le programme

 CursorOn(SauveCurs) { Juste avant la fin}

end.
```

- La routine suivante sera utilisée dans un programme assembleur pour modifier la position courante du curseur:

```
GotoXY   PROC      NEAR
;=========================================;
; Cette procédure positionne le curseur  ;
; à la colonne AL et à la ligne AH       ;
;                                        ;
; Paramètres d'entrée   : AL,AH          ;
; Paramètres de sortie  : Aucun          ;
; Registres détruits    : Aucun          ;
;=========================================;
         PUSH BX   ;Préserve les registres
         PUSH DX

         MOV DX,AX ;DH <-- AH (ligne)
                   ;DL <-- AL (colonne)
         MOV AH,2  ;Fonction 02h de
                   ;l'interruption 10h
         MOV BH,0  ;Page 0
         INT 10h   ;Appel de l'interruption

         POP DX    ;Récupère les registres
         POP BX
         RET
GotoXY   ENDP
```

L'interruption 14h — Gestion du port série.

L'interruption 14h permet de gérer le port série auquel on pourra notamment connecter un modem. Notons que pour des communications dont la vitesse doit dépasser 9600 bauds, il faudra réécrire une nouvelle routine.

Elle dispose des 4 fonctions suivantes :

00h Initialisation des paramètres du port série (vitesse, parité, bits stop, taille d'un caractère).

01h Envoie un caractère (émission).

02h Lecture d'un caractère sur le port série (réception).

03h Lecture de l'état du port série.

Avant l'appel à la fonction 00h, il faudra initialiser le registre DX avec le numéro du port série (à partir de 0) et AL comme suit :

- Les bits 6,7,8 contiendront la vitesse, les codes possibles sont :

  - **000** 110 bauds
  - **001** 150 bauds

- **010** 300 bauds
- **011** 600 bauds
- **100** 1200 bauds
- **101** 2400 bauds
- **110** 4800 bauds
- **111** 9600 bauds

- Les bits 3 et 4 contiendront la parité, les codes possibles sont :

  - **00** Pas de parité
  - **01** Parité impaire
  - **10** Pas de parité
  - **11** Parité paire

- Le bit 2 contiendra le nombre de bits "stop", les codes possibles sont :

  - **0** Un bit "stop"
  - **1** Deux bits "stop"

- Les bits 0 et 1 contiendront la taille des caractères, les codes possibles sont :

  - **00** Pas utilisé
  - **01** Pas utilisé
  - **02** Caractères sur 7 bits
  - **03** Caractères sur 8 bits

Exemples de programmation

Le programme suivant contient quelques routines de base pour un programme de communication.

```
Program RS232;

Type
 Str255-String[255];
 RegType = Record
  Case Integer Of
   1:(AX,BX,CX,DX,BP,SI,DI,DS,ES: Integer);
   2:(AL,AH,BL,BH,CL,CH,DL,DH:Byte);
  end;

Var
Reg:Regtype;
MaxPorts:byte;

Function NbrRs232:Byte;

 {Cette fonction doit être appelée en      }
 {premier lieu pour connaître le nombre    }
 {de RS232 installés                       }
var
 temp:byte;

begin
 NbrRs232:=0;
 Intr($11,reg);
 Temp:=(Reg.ax and $0E00) shr 9;
 If Temp>2 Then temp:=2;
 NbrRs232:=temp;
end;{NbrRs232}

Function InitRs232( Port    :Byte;
                    Vit     :Integer;
                    Parit   :Char;
                    Donnee  :Byte;
                    Stop    :byte):Boolean;
{Cette fonction initialise le port RS232     }
```

```
begin

 With Reg Do
 begin
  InitRs232:=False;
  If (Port<1) or (Port>MaxPorts) Then Exit;
  ax:=$0000;
  dx:=Port-1; {Ports numérotés à partir de 0}
  Case Vit of
   110   :Ax:=Ax Or 0;
   150   :Ax:=Ax Or 32;
   300   :Ax:=Ax Or 64;
   600   :Ax:=Ax Or 96;
   1200  :Ax:=Ax Or 128;
   2400  :Ax:=Ax Or 160;
   4800  :Ax:=Ax Or 192;
   9600  :Ax:=Ax Or 224;
  Else
   Exit;{il y a erreur}
  End;{Case vit}

  Case Parit Of
   'N','n':ax:=ax or 0;
   'O','o':ax:=ax or 8;
   'E','e':ax:=ax or 24;
  Else
   Exit;{erreur de parité}
  End;{case parit}

  Case Donnee OF
   7:Ax:=ax or 2;
   8:ax;:ax or 3;
  Else
   Exit;{erreur sur le nombre de bits de donnée}
  End;{case donnée}

  Case Stop OF
   1:Ax:=ax or 0;
   2:ax;:ax or 4;
  Else
   Exit;{erreur sur le nombre de bits stop}
  End;{case stop}

 end;{with reg}
Intr($14,reg);
InitRs232:=True;
```

```
end;{InitRs232}

Function Rs232Ready(Port:Byte);

begin

  Rs232Ready:=False;
  If (Port<1) or (Port>2) Then Exit;
  With Reg Do
  begin
   ax:=$0300;
   dx:=port-1;
   Intr($14,reg);
   If (Al and $0300)=0 Then
    exit; {00110000}
   Rs232Ready:=True;
  End;{with reg}

end;{Rs232Ready}

Function Envoie(Port:byte;st:str255):boolean;

  {Envoie une chaîne de caractères au modem}

  Var
  k,l:byte;

begin
  Envoie:=false;
  With Reg Do
  begin
   Ah:=$01;
   dx:=port-1;
   For k:=1 to length(st) do
   begin
    l:=0;
    {attend un laps de temps si le port n'est   }
    {pas prêt                                   }
    While (l<250) And (not Rs232Ready(port)) do
     l:=l+1;
    If l=>250 then exit;

    Al:=ord(st[k]);
```

```
    Int($14,reg);
    If (ah and $8E)<>0 Then
     Exit;{il y a erreur}
   end;{for}
  End;{with}
  Envoie:=true;
end;{envoie}

Function Compose(Port:Byte;Tel:str255);

  {Cette fonction compose le numéro de      }
  {téléphone pour un modem hayes ou         }
  {compatible                               }

begin
  Compose:=false;
  If Not(Envoie(Port,'ATDT'+Tel) Then
   Exit;
  Compose:=true;
end;{Compose}

begin {programme}

  MaxPorts:=NbrRs232;

   {initialise com1 à 1200 bauds, parité paire,   }
   {7 bits de donnée, 1 bit stop                  }
  If not InitRs232(1,1200,'E',7,1) then
   halt;{erreur d'initialisation --> stop}

  Compose('00,,14155311820'); {compose ce numéro}
  Envoie('Bonjour');{envoie cette chaîne de caract}
end.{Programme}
```

L'interruption 16h —Gestion du clavier.

Cette interruption s'occupe de la gestion du clavier, les caractères sont lus dans le buffer du DOS et transmis au programme appelant. Elle dispose des 3 fonctions suivantes :

00h Lecture d'un caractère.

01h Teste si un caractère est disponible.

02h Lecture de l'état des touches SHIFT, ALT, CTRL, CAPSLOCK, NUMLOCK, SCROLL-LOCK et INSERT.

Il faut aussi savoir que lorsqu'on enfonce une touche, l'interruption **16h** renvoie deux informations :
- Le **code ascii** du caractère (dans AL).
- Le "**Scan code**" de la touche (dans AH).

Le "scan code" est la **position** de la touche sur le clavier.

Ainsi sur un clavier AZERTY, le scan code de la touche "a" est 16, et le scan code du "q" sur un clavier QWERTY est aussi 16 puisque la position de la touche est identique sur les deux types de claviers. ***Les "scan codes" sont nécessaires pour identifier des touches qui n'ont pas de code ascii*** (touches de fonction, séquences ALT-touche).

Exemples de programmation

- **La routine suivante appelle la fonction 02h de l'interruption 16h pour connaître l'état des touches de contrôle du clavier :**

```
Type
 RegType = Record
  Case Integer Of
  1:(AX,BX,CX,DX,BP,SI,DI,DS,ES: Integer);
  2:(AL,AH,BL,BH,CL,CH,DL,DH:Byte);
 end;

Var
 Reg:RegType;
 Last:Byte;    {Pour n'afficher que quand il y }
               {a une modification d'état}

Procedure EtatKbd;

begin
  Reg.ah:=$02;
  Intr($16,reg);
  If Reg.al<>Last then
  begin
   If (Reg.al and $01) <>0 then
        Writeln('Shift Droit Actif');
   If (Reg.al and $02) <>0 then
        Writeln('Shift Gauche Actif');
   If (Reg.al and $04) <>0 then
        Writeln('Ctrl Actif');
   If (Reg.al and $08) <>0 then
        Writeln('Alt Actif');
   If (Reg.al and $10) <>0 then
        Writeln('Scroll Lock Actif');
   If (Reg.al and $20) <>0 then
        Writeln('Num Lock Actif');
   If (Reg.al and $40) <>0 then
        Writeln('Caps Lock Actif');
   If (Reg.al and $80) <>0 then
        Writeln('Insert Actif');
   Writeln;
```

```
    Last:=Reg.al;
   end;{if reg.al}
end;{EtatKbd}

begin
{Appelle cette routine sans fin pour}
{permettre de combiner les différentes}
{touches}

Last:=255;
While True Do
 EtatKbd;

end.
```

- Voici une routine utilisant les fonctions **00h** et **01h** de l'interruption **16h**. Elle peut servir pour la lecture des caractères en provenance du clavier et peut même exécuter une tâche en "background" si aucun caractère n'est frappé :

```
Function ReadChar:Char;
Var
 Touche:Boolean;
Begin
 Touche:=false;
 While not(Touche) Do
 begin
  Reg.ah:=$01;      {Regarde si un caractère }
                    { est présent           }
  Intr($16,reg);    {appelle l'interruption}
  If Boolean(Reg.flags and $40) then
  {Pas de caractères, appelle une tâche}
  {à exécuter en "background"}
  begin
   {insérer ici un appel à une ou plusieurs }
   {procédures qui s'exécutent "vite" comme }
   {par exemple:                            }
   {- lire et afficher l'heure à l'écran    }
   {- imprimer quelques caractères          }
```

```
      {- ...                                    }
     end
     else
     begin
      Ah:=0;
      intr($16,reg); {lit le caractère}
      Touche:=true;
      ch:=Char(Reg.al);
      writeln(ch),{affiche le caractère}
      ReadChar:=Ch;
     end;
    end;{while not(touche) do}
  End;{ReadChar}
```

- Si on utilise les fonctions définies ci-dessus pour la gestion du clavier, il est utile d'écrire un petit programme qui affichera pour chaque touche enfoncée son code ascii ainsi que son scan code. La fonction **00h** de l'interruption **16h** renvoie :

  - Dans **AL** le code ascii du caractère frappé.
  - Dans **AH** le "scan code" de la touche.

```
Program scancodes

 Type
  RegType = Record
   Case Integer Of
   1:(AX,BX,CX,DX,BP,SI,DI,DS,ES: Integer);
   2:(AL,AH,BL,BH,CL,CH,DL,DH:Byte);
  end;

 Var
  Reg:RegType;
  Last:Byte;    {Pour n'afficher que quand il y }
                {a une modification d'état}
 chh:char;


begin {program}
  chh:=chr(0); {initialise chh}
  While chh<>^A Do
```

```
  {le programme s'arrête quand on frappe ^A}
   With reg do
   begin
    ah:=$01;
    intr($16,reg);
    if boolean(flags and $40) then
    begin
     {pas de caractère frappé}
    end
    else
    begin
     ah:=0;
     intr($16,reg); {lit le caractère}
     chh:=char(al);
      {le code ascii est renvoyé dans AL  }
      {et le scan code dans ah)           }
     writeln('Code Ascii:',al,' Scan code:',ah);
    end;
   end;{with/while}
end.{program}
```

L'interruption 17h — Gestion de l'imprimante.

Elle dispose des 3 fonctions suivantes :

00h Envoi d'un caractère sur l'imprimante.
01h Initialise l'imprimante.
02h Lecture de l'état de l'imprimante (prête,...).

Il est conseillé d'appeler la fonction **02h** avant la fonction **00h** pour s'assurer que l'imprimante est prête. La fonction **02h** renvoie alors dans le registre AH un des codes suivants :

- **00h** Pas de réponse
- **01h** Pas utilisé
- **04h** Pas utilisé
- **08h** Erreur
- **10h** Imprimante sélectionnée
- **20h** Plus de papier
- **80h** Imprimante prête

Ainsi AH=144 indique que les deux codes **10h** (imprimante sélectionnée) et **80h** (imprimante prête) ont été renvoyés; l'imprimante est donc prête à recevoir un caractère.

Exemples de programmation

- **La fonction suivante teste si l'imprimante est prête :**

```
Function PrinterReady:Boolean;
begin
 Reg.ax:=$0200;       {Fonction 02h}
 Reg.dx:=0;     {imprimante 0}
 Intr($17,reg);       {appelle l'interruption 17h}

{La fonction renvoie dans AH les informations}
{suivantes: bit  signification                }
{           ---  --------------               }
{           0    Timeout                     }
{           1,2  Pas utilisés                }
{           3    Erreur                      }
{           4    Imprimante sélectionnée     }
{           5    Plus de papier              }
{           6                                }
{           7    Imprimante prête            }

 If Reg.ah=144 Then {bit 4 et 7 sont à 1}
  PrinterReady:=True
 Else
  PrinterReady:=False;

end;{PrinterReady}
```

- La fonction suivante imprime une chaîne de caractères :

```
Type
 str255=string[255];

Function PrintString(s:str255):boolean;
Var i,j:integer;
begin
```

```
 i:=length(s);
 j:=1;
 while j<=i do
 begin
  if printerReady then
  {teste si l'imprimante est prête avant chaque}
  {caractère}
  begin
   reg.ah:=0;
   reg.al:=ord(s[j]);
   intr($17,reg);
  end
  else {erreur, l'impriante n'est plus prête}
  begin
   PrintString:=false; {signale l'erreur}
   exit;{abandonne l'impression}
  end;
  j:=j+1;
 end;{while}
PrintString:=true; {tout s'est bien passé}
end;{PrintString}
```

Les fonctions 3Fh et 40h de l'interruption 21h — Lecture/Ecriture sur un périphérique.

Comme nous le verrons, ces deux fonctions seront utilisées pour lire/écrire des informations à partir d'un fichier. Depuis les versions 2.xx de MS-DOS, les fichiers sont gérés par l'intermédiaire de "**handles**". Les "handles" sont des numéros qu'il faut spécifier avant toute opération sur un fichier. Les numéros suivants sont réservés pour les périphériques :

- **0** Clavier.
- **1** Ecran.
- **2** Ecran.
- **3** Port série (modem, souris).
- **4** Imprimante.

Ces fonctions doivent spécifier comme paramètres les valeurs suivantes :

- Le numéro de la fonction.
- Le numéro du "handle".
- Le nombre de bytes à lire (3Fh) ou écrire (40h).
- L'adresse d'une zone mémoire qui recevra (3Fh)/ contiendra (40h) les données à lire/écrire.

- **Au retour de la fonction 3Fh, l'indicateur CARRY indique le résultat de l'opération :**

  - **Si l'indicateur Carry est à 0, AX contient le nombre de bytes effectivement lus et DS:DX l'adresse de la zone mémoire.**

  - **Si l'indicateur Carry est à 1, il y a eu erreur et AX contient un des codes suivants :**

    05h Accès refusé.
    06h "Handle" inexistant ou incorrect.

- Au retour de la fonction **40h**, l'indicateur CARRY indique le résultat de l'opération :

  - Si l'indicateur Carry est à 0, AX contiendra le nombre de bytes effectivement écrits.

  - Si l'indicateur Carry est à 1, il y a eu erreur et AX contient un des codes suivants :

    05h Accès refusé.
    06h "Handle" inexistant ou incorrect.

Exemple de programmation

Le programme suivant convertit des données majuscules en minuscules. En voici quelques exemples d'utilisation ("convert" est le nom du programme) :

- Pour afficher les fichiers d'un répertoire en lettres minuscules :

```
Dir :convert
```

- Pour afficher le contenu d'un fichier ABC.DAT en minuscules :

```
TYPE ABC.DAT :convert
```

- Pour imprimer en minuscules le contenu d'un fichier :

```
TYPE ABC.DAT :convert >lpt1
```

- Pour convertir un fichier Maj.dat en min.dat :

```
Convert  <MAJ.DAT >MIN.DAT
```

Notes :

- Le ':' est le caractère ascii 124 (utilisé dans la création des filtres).

- Dans le dernier exemple, le '<' indique que les données à lire sur le périphérique d'entrée standard doivent l'être dans le fichier MAJ.DAT et '>' indique que les données à écrire dans le périphérique de sortie sont à écrire dans MIN.DAT (redirection de périphérique).

  Pour en savoir plus concernant les filtres et le reroutage, voir *MS-DOS facile* (MS n° 843).

```
Program convert;

 Type
  RegType = Record
   Case Integer Of
    1:(AX,BX,CX,DX,BP,SI,DI,DS,ES: Integer);
    2:(AL,AH,BL,BH,CL,CH,DL,DH:Byte);
   end;

 Var
  Reg:RegType;
  Buffer:array[1..$80] of char;
  bytes:integer;
  fin:boolean;
  i,j,errcode:integer;

 Function Lecture:Boolean;


 begin
  With Reg Do
  begin
   ax:=$3F00;
   bx:=$0000;
   cx:=$0080;
   ds:=dseg;
   dx:=ofs(buffer);
   intr($21,Reg);
   if ((Flags and 1) <> 0) then
```

```
      {teste flag CARRY  }
      ErrCode := Ax
     else
      begin
       ErrCode := 0;
       Bytes:=ax;
       if bytes=0 then
        fin:=true;
      end;
     If errcode <>0 then
      Lecture:=false
     else
      Lecture:=true;
    end;{with}
 end;{Lecture}

 Procedure Ecriture;

 begin
    With Reg Do
    begin
     ax:=$4000;
     bx:=$0001;
     cx:=$Bytes;
     ds:=dseg;
     dx:=ofs(buffer);
     intr($21,Reg);
     if ((Flags and 1) <> 0) then
      {teste flag CARRY  }
      ErrCode := Ax
     else
      ErrCode := 0;
    end;{with}
 end;{Ecriture}

 Procedure Transforme;

 begin
    {convertit le contenu du "buffer" en minuscules}
   for i:=1 to bytes do
    if buffer[i] in ['A'..'Z'] then
     buffer[i]:=chr(ord(buffer[i])+32);
```

```
end;{transforme}

 begin {program}
  fin:=false;
  While not(fin) do
   begin
    fillchar(buffer,sizeof(buffer),0);
    If Lecture Then
    begin
     transforme;
     Ecriture;
    end;'if lecture}
    writeln;
   end;{while};
  writeln('Opération terminée');
end. {program}
```

Note : Une extension intéressante de cette application consisterait à rendre *résident* un programme de ce type dont le travail serait de capturer les informations envoyées à l'imprimante (par les autres programmes) et à les rediriger dans un fichier.

3. Les périphériques "bloc"

Ces périphériques peuvent effectuer des opérations d'E/S (entrées/sorties) avec des blocs (ensembles) de caractères. Le bloc de caractères est contenu dans une zone mémoire, l'adresse et la longueur de la zone seront transmises comme paramètres à l'interruption qui s'occupe de la gestion du périphérique concerné par l'opération. Ces périphériques ne possèdent pas de nom (à l'inverse des périphériques "caractères"). Les disques, par exemple, sont des périphériques "bloc", ils peuvent être divisés en plusieurs unités, à chaque unité, on associera une lettre (A, B, etc.). Le programme de contrôle d'un périphérique (device driver) pourra être responsable de plusieurs unités, la séquence d'attribution des lettres aux différentes unités est déterminée par la séquence des différents "drivers" dans la chaîne.

L'interruption 13h — Gestion des disques.

L'interruption 13h s'occupe de la gestion de toutes les unités de disques, elle propose notamment les services suivants :

00h Initialisation du disque.
01h Lecture du résultat de la dernière opération.
02h Lecture d'un secteur (bloc de caractères).
03h Ecriture d'un secteur (bloc de caractères).
04h Vérification d'un secteur.
08h Lecture des paramètres du disque.
10h Teste si le disque est prêt.
0Ch Positionnement des têtes de lecture.

Notons qu'il est conseillé de réinitialiser le disque (fonction **00h**) avant chaque opération. Une explication détaillée de la structure du disque, des secteurs et des techniques d'allocation d'espace disque à un fichier pourra être trouvée au chapitre 6.

Exemple de programmation

Le programme ci-dessous permet de "parquer" les têtes du disque dur avant d'éteindre l'ordinateur. Cette action est recommandée même si on ne compte pas déplacer la machine. Certaines études auraient en effet montré que lors du démarrage il est conseillé de ne pas avoir les têtes de lecture n'importe où car cela risque d'endommager (à terme) certaines zones du disque.

```
Program Parque;

Type
 RegType = Record
  Case Integer Of
   1:(AX,BX,CX,DX,BP,SI,DI,DS,ES: Integer);
   2:(AL,AH,BL,BH,CL,CH,DL,DH:Byte);
  end;

Var
 Reg:RegType;
 disque,nombre:byte;
 ErrCode:Integer;

begin
 ErrCode:=0;
 With reg do
 begin
    {La fonction 08h renvoie les paramètres    }
    {d'un disque ainsi que le nombre           }
    {de disques installés                      }
  ah:=$08;
  dl:=$80
  intr($13,reg);
    {Au retour,  cx contient le nombre         }
    {       de cylindres}
    {       dl contient le nombre de disques   }
  Nombre:=dl;
```

```
    {parque tous les disques existants:          }
   For disque:=0 to (nombre-1) do
   begin
    ah:=$0C;  {positionne la tête de lecture     }
              {sur le dernier cylindre+1,        }
    al:=$01;  {secteur 1                         }
    cx:=cx+1;
    dl:=$80+disque;
    intr($13,reg);
    if ((Flags and 1) <> 0) then
      {teste flag CARRY                          }
     ErrCode :=ErrCode+Ax;

      {Lit les paramètres du disque suivant      }
    ah:=$08;
    dl:=$80+k;
    intr($13,reg);
     {Au retour,  cx contient le nombre          }
     {            de cylindres                   }
  end;{for}

  If ErrCode<>0 Then
   Writeln('Erreur de parquage...')
  Else
  begin
   writeln;
   Writeln('Têtes parquées avec succès...');
   writeln;
   Writeln('Eteignez l'ordinateur ou tapez
    une touche pour reprendre la session.');
   Repeat until keypressed;
  end;
 end;{with reg}
end.{parque}
```

4. Autres périphériques

Parmi les périphériques additionnels que l'on peut connecter à un ordinateur, on peut citer :

- La souris.
- Un joystick.
- Un "light pen".

Certaines fonctions de l'interruption **33h** sont dédiées à la gestion de la souris.

L'interruption 33h — Gestion de la souris.

Elle dispose des fonctions suivantes :

- **00h** Test de présence de la souris.
- **01h** Montre le pointeur de la souris.
- **02h** Cache le pointeur de la souris.
- **03h** Lecture de la position de la souris et de l'état des boutons.
- **04h** Modification de la position de la souris.
- **05h** Lecture des boutons enfoncés.
- **06h** Lecture des boutons relâchés.
- **07h** Modification de l'intervalle de déplacement horizontal.
- **08h** Modification de l'intervalle de déplacement vertical.
- **09h** Définition de curseur graphique.

- **0Ah** Définition de curseur en mode texte.

Avec les variables suivantes, nous allons définir les procédures essentielles de gestion de la souris.

```
Type
 RegType = Record
  Case Integer Of
  1:(AX,BX,CX,DX,BP,SI,DI,DS,ES: Integer);
  2:(AL,AH,BL,BH,CL,CH,DL,DH:Byte);
 end;

Var
 Regs:RegType;
```

Les fonctions de status et de gestion du pointeur de la souris

La procédure "MouseStat" utilise la fonction 00h de l'interruption 33h pour vérifier la présence de la souris et connaître le nombre de boutons disponibles.

```
Procedure MouseStat(Var Status,NbBout:Integer);
  {Renvoie:  Status=0   si la souris n'est pas  }
  {                     installée               }
  {          Status=-1  autrement               }
  {                                             }
  {          NbBout=    Le nombre de boutons    }
  {                     de la souris            }

 Begin
  With Regs Do
   Begin
    Ax:=0; {Fonction 00h de l'interruption 33h}
    Intr($33,Regs);
    Status:=Ax;
    NbBout:=Bx;
   End;
 end;{MouseStat}
```

Les procédures "MontrePointeur" et "CachePointeur" utilisent respectivement les fonctions **01h** et **02h** de l'interruption **33h**. Elle permettent d'afficher ou de supprimer le pointeur de la souris.

```
Procedure MontrePointeur;

begin
  With Regs Do
   Begin
    Ax:=1; {Fonction 01h de l'interruption 33h}
    Intr($33,Regs);
   End;
end;{MontrePointeur}
```

```
Procedure CachePointeur;

begin
  With Regs Do
    Begin
     Ax:=2; {Fonction 02h de l'interruption 33h}
     Intr($33,Regs);
    End;
end;{CachePointeur}
```

Les fonctions de positionnement de la souris

La procédure "PosEtBout" utilise la fonction 03h de l'interruption 33h pour connaître la position de la souris ainsi que l'état des boutons.

```
Procedure PosEtBout(Var x,y,bout:Integer);

  {Retourne:                                          }
  {                                                   }
  {      -dans X et Y la position de la               }
  {       souris                                      }
  {      -dans BOUT l'état des boutons:               }
  {        =1 indique que le gauche est enfoncé       }
  {        =2 indique que le droit est enfoncé        }
  {        =4 indique que le milieu (s'il existe)     }
  {           est enfoncé                             }
  {        =3 indique que le gauche ET le droit       }
  {           sont enfoncés (3=1+2)                   }
  {        etc.                                       }

begin
 With Regs Do
  Begin
   Ax:=3; {Fonction 03h de l'interruption 33h}
   Intr($33,Regs);
   x:=CX;
   y:=DX;
   Bout:=BX;
  End;
end;{PosEtBout}
```

La fonction "ModifPos" utilise la fonction **04h** de l'interruption **33h** pour modifier la position de la souris.

```
Procedure ModifPos(x,y:Integer);

{x=colonne, y=ligne}

begin
 With Regs Do
  Begin
   Ax:=4; {Fonction 04h de l'interruption 33h}
   CX:=x;
   DX:=y;
   Intr($33,Regs);
  End;
end;{ModifPos}
```

La procédure "BoutEnfonce" utilise la fonction **05h** de l'interruption **33h** pour obtenir le nombre de fois qu'un bouton a été enfoncé ainsi que l'état général ACTUEL de tous les boutons. Il faudra spécifier, comme paramètre d'entrée le bouton pour lequel on souhaite l'information. Au retour, la fonction renverra les informations suivantes:

- **Status** contient l'état actuel des boutons de la souris (cf. fonction 03h ci-dessus).

- **Compt** indique le nombre de fois que le bouton a été enfoncé (ce compteur est remis à 0 après l'appel à la fonction 05h).

- **Col** indique la colonne du pointeur lors du dernier enfoncement du bouton spécifié à l'appel.

- **Ligne** indique la ligne du pointeur lors du dernier enfoncement du bouton spécifié à l'appel.

```
ocedure BoutEnfonce(   Bouton:integer;
                       Var Status,
                       Compt,
                       Horz,
                       Vert:Integer);

gin
ith Regs Do
egin
 Ax:=5;
 Bx:=Bouton;
 Intr($33,Regs);
 Status:=Ax;
 Compt:=Bx;
 Horz:=Cx;
 Vert:=Dx;
end;
nd;{BoutEnfoncé}
```

pourra écrire une autre procédure "BoutRelâche", ntique à la précédente à l'exception près qu'elle pelle la fonction **06h** de l'interruption **33h** au lieu la fonction **05h**. Cette procédure renverra le mbre de fois qu'un bouton a été relâché.

Modification de l'aire de déplacement de la souris

Les procédures "MaxHorz" et "MaxVert" utilisent respectivement les fonctions 07h et 08h de l'interruption 33h pour limiter l'aire de déplacement du pointeur de la souris.

```
Procedure MaxHorz(Min,Max:Integer);
  {Les valeurs doivent être comprises entre  }
  {0 et 639                                  }
begin
  With Regs Do
    Begin
      Ax:=07; {Fonction 07h de l'interruption 33h}
      CX:=Min;
      DX:=Max;
      Intr($33,Regs);
    End;
end;{MaxHorz}

Procedure MaxVert(Min,Max:Integer);
  {Les valeurs doivent être comprises entre  }
  {0 et 199                                  }
begin
  With Regs Do
    Begin
      Ax:=08; {Fonction 08h de l'interruption 33h}
      CX:=Min;
      DX:=Max;
      Intr($33,Regs);
    End;
end;{MaxVert}
```

Exemple de programmation

Le programme suivant permet d'utiliser le pointeur de la souris en mode "texte" ou "graphique" (cga 640*200 et pas Hercules). Dans le premier cas, on l'utilisera pour positionner le curseur dans un texte et dans le second pour dessiner.

```
Program TestMouse1;

{   INSERER ICI LES PROCEDURES QUE NOUS   }
{   VENONS DE DEFINIR                     }

Var

 x1,y1,x2,y2:Integer;
 NBut,Stat:Integer;
 Fin,TextOn:Boolean;
 x,y,bout:integer;
 Chh:Char:

Procedure ChangePointeur(taille:integer);
 {définition du curseur en mode texte}
begin
 reg.ax:=$0A;
 reg.cx:=0;
 reg.dx:=taille;
 Intr($33,Reg);
end;

Procedure ModeText;
begin
 textOn:=True;
 TextMode;{passe en mode 80*25}
 GotoXy(1,1);
 Writeln(' TEXTES       *');
 writeln('              *');
 writeln(' *************');
 ChangePointeur(15);
 MouseStat(stat,Nbut);       {initialise}
```

```
 MontrePointeur;
end;{modetext}


Procedure ModeGraph;
begin
 TextOn:=false;
 Hires;{passe en haute résolution 640 * 200}
 GotoXy(1,1);
 Writeln(' GRAPHIQUES  *');
 writeln('             *');
 writeln(' *************');
  {récupère le curseur d'origine qui est une }
  {flèche en haute résolution                }
 MouseStat(stat,Nbut);   {initialise}
 MontrePointeur;
end;{modegraph}

begin{programme}

 Clrscr;
 Writeln('Programme de démonstration');
 writeln;
 writeln('En mode "texte", vous pouvez:');
 writeln;
 writeln(' -   Positionner le pointeur avec la
               souris');
 writeln(' -   Cliquer le bouton gauche pour
               "ancrer" le curseur');
 writeln(' -   Taper du texte à ce point
               d'ancrage');
 writeln;
 writeln('En mode "graphique", vous pouvez:');
 writeln;
 writeln(' -    Positionner le pointeur avec la
                souris');
 writeln(' -    Cliquer le bouton gauche pour
                "ancrer" le curseur');
 writeln(' -    Maintenir le bouton enfoncé et
                tracer des dessins');
 writeln;
 writeln('Pour quitter, positionner le curseur
          dans le coin supérieur gauche de
          l'écran);;

 Fin:=false;
```

```
x1:=-1;
x2:=0;
y1:=-1;
y2:=0;
MouseStat(stat,Nbut);   {initialise}
If stat=0 then halt;
ModeText;               {passe en mode Texte}

While Not(Fin) Do
begin
 if texton then
   if keypressed then
    begin
     read(kbd,chh);
     write(chh);
    end;{if keypressed}

 PosEtBout(x,y,bout);

 if (x<80) and (y<20) then
  {Si le curseur est dans le coin supérieur }
  {gauche de l'écran, on arrête le programme}
 begin
  write(^G);
  fin:=true;
 end
 else
 begin
  if bout=2 then    {bouton droit cliqué      }
                    {On change de mode  }
  begin
    If textOn then
     ModeGraph
    else
     modetext;
  end
  else
    if bout=1 then {bouton gauche cliqué}
    begin
     If TextOn then
     {on est en mode texte, on positionne le
     {curseur en convertissant X (actuellement
     {exprimé entre 1 et 640) en une valeur
     {comprise entre 0 et 80 et, Y sera
```

```
        {converti en une valeur comprise entre 1    }
        {et 25, actuellement c'est une valeur entre}
        {1 et 200                                   }
          Gotoxy(((x+1) div 8)+1,(y div 8)+1)
        Else
         begin
          if x1=-1 then {premier clic}
          begin
           x1:=0;
           x2:=x;
           y2:=y;
           Plot(x,y,1);{affiche un point}
          end
          else {autres clics}
          begin
           x1:=x2;
           y1:=y2;
           x2:=x;
           y2:=y;
           Draw(x1,y1,x2,y2,1); {trace une ligne}
          end;{if x1=-1 else}
        end;{if texton else}
      end;{if bout=1}
    end;{else}

  End;{While not fin}
  TextMode;
  ClrScr;
end.{TestMouse1}
```

Le programme suivant permet de tester si l'utilisateur a cliqué une fois ou deux. Nous utiliserons la fonction 2Ch de l'interruption 21h pour lire l'heure du système et ainsi calculer un délai maximum pour que deux pressions successives sur le bouton gauche de la souris soient considérées comme un "double clic".

```
Program testMouse2;

Var
  Start,Stop:Real;
  xr:Byte;
  Maxdelai:real;
  x,y,Bout:integer;

Procedure Time(var sec:real);
{renvoie l'heure actuelle exprimée en secondes}

begin
 with reg do
 begin
   ah:=$2C;
   intr($21,reg);
   sec:=1.0*(Dh+60.0*(Cl+60.0*Ch)+Dl/100.0);
 end;{with}
end;{time}

Function Clics:Byte;
begin
 clics:=0;

  {met en route le chronomètre}
 Time(start);
 stop:=start;

  {attend que l'utilisateur ait lâché le bouton}
 While (bout=1) DO
  PosEtBout(x,y,bout);
```

```
    {contrôle, délai passé???}
   Time(stop);

     {on lui donne "maxdelai" secondes pour    }
     {cliquer une seconde fois                 }
   While ((stop-start)<Maxdelai) do
   begin
    PosEtBout(x,y,bout);
    If bout=2 then exit;{il a poussé le droit}
    if bout=1 then {ok, le gauche}
    begin
      {attend qu'il le lâche maintenant:}
     while (bout=1) do
     PosEtBout(x,y,bout);
     clics:=1;
     exit;{plus besoin de continuer à attendre}
    end;{if bout=1}
    time(stop);{contrôle du timing}
  end;{while}
 end;{clics}


 begin{testmouse2}

    {accorde un délai max de 0.4 secondes entre }
    {les deux clics                             }
  Maxdelai:= 0.4;
  Mousestat(x,bout);
  xr:=0;
  bout:=0;

    {pour arrêter, cliquer le bouton droit
  while (bout<>2) do
  begin
   PosEtBout(x,y,bout);
   if bout=1 then {gauche cliqué}
   begin
    xr:=clics;
    if xr>0 then
     writeln('DOUBLE CLIC')
    else
     writeln('SIMPLE CLIC');
   end;{if bout=1}
  end;{while}
End.
```

5. L'interruption 24h

Tout le monde connaît bien le fameux message:

```
Not error....
Abort, retry, ignore ?
```

Il apparaît quand l'imprimante n'est pas prête, quand la porte du lecteur n'est pas fermée, etc.

Ce message est envoyé par l'interruption **24h** qui gère **tous** les cas d'erreur critique.

Cependant, ce message est fort ennuyeux pour les programmes car il décale tout l'écran (ce qui n'est pas très esthétique). De plus, si l'utilisateur répond "Abort", le programme sera interrompu immédiatement alors qu'il est peut-être en plein milieu d'une opération importante!

Comme pour toutes les interruptions, MS-DOS ne voit aucun inconvénient à ce que nous réécrivions cette routine et que nous l'installions à la place de la sienne.

Pour cela, nous devons savoir que :

Lorsque l'interruption **24h** reçoit le contrôle, le code d'erreur se trouve dans le registre **DI.**

Les codes d'erreur possibles sont :

1 **Tentative d'écriture sur un disque protégé contre l'écriture.**
2 **Unité inconnue.**
3 **Disque pas prêt (porte du drive ouverte).**
4 **Commande Inconnue.**
5 **Erreur de donnée (CRC).**
6 **Erreur dans la longueur de la structure de la requête.**
7 **Erreur de positionnement.**
8 **Type de média inconnu.**
9 **Secteur pas trouvé.**
10 **Plus de papier dans l'imprimante.**
11 **Erreur d'écriture.**
12 **Erreur de lecture.**
13 **Erreur générale (disquette pas formatée, ...).**

Il faut renvoyer dans AL une des valeurs suivante :

00h Indique qu'il faut ignorer l'opération.
01h Indique qu'il faut réessayer.
02h Indique qu'il faut terminer le programme.

Exemple de programmation

Nous allons maintenant réécrire l'interruption 24h et ainsi résoudre le problème du "Abort Retry Ignore". Lorsque notre interruption reçoit le code d'erreur dans le registre DI, elle le stocke dans une constante avec TYPE ("typed constant") du programme Pascal. Nous renvoyons ensuite le code 0 (dans AL) à MS-DOS afin de lui commander d'ignorer l'erreur. Dorénavant, après chaque opération d'entrée/sortie, on n'appellera plus la fonction prédéfinie "IORESULT", mais une fonction personnelle que nous avons créée : "INT24result". Cette fonction teste la constante et le code d'erreur éventuel de Turbo-Pascal. S'il n'y a pas d'erreur, cette fonction renverra 0.

```
Program Int24demo;
Const
  Int24ErrCode: byte=0;
   {contient le code d'erreur      }
   {initialisé par INt24...        }

  OldInt24OFs : integer=0;
  {sauve l'adr              }
  {de l'ancienne int24h}
  OldInt24Seg : integer=0;

var
 err:integer;
 fichier:text;
 Nom:string[64];

Function Int24Result:Integer;
 {Renvoie le code d'erreur Int24 et          }
 {turbo (ioresult)                           }
 {cette fonction combine un éventuel code    }
```

```
  {d'erreur critique avec la fonction IORESULT }
  {de TURBO, elle peut donc être utilisée      }
  {systématiquement à la place de IORESULT     }

  { Les codes pour int24 sont:                 }
  {                                            }
  {   1.  Tentative d'écriture sur un disque   }
  {       protégé contre l'écriture            }
  {   2.  Unité inconnue                       }
  {   3.  Disque pas prêt (porte du drive ouverte) }
  {   4.  Commande Inconnue                    }
  {   5.  Erreur de donnée (CRC)               }
  {   6.  Erreur dans la longueur de la        }
  {       structure de la requête              }
  {   7.  Erreur de positionnement             }
  {   8.  Type de média inconnu                }
  {   9.  Secteur pas trouvé                   }
  {   10. Plus de papier dans l'imprimante     }
  {   11. Erreur d'écriture                    }
  {   12. Erreur de lecture                    }
  {   13. Erreur générale (disquette pas       }
  {       formatée,...)                        }

begin
 Int24Result:=Ioresult or (int24ErrCode shl 8);
 Int24ErrCode:=0;
end;


Procedure Int24;
  {Notre nouvelle routine, appelée par    }
  {MS-DOS en cas d'erreur critique...     }
begin
  Inline(
  $89/$F8/   {Mov ax,di  ;ax<--code d'erreur      }
  $FE/$C0/   {Inc al     ;Nos codes commencent à 1 }
             {           ;ceux de MS-DOS à 0...   }
  $2E/       {cs:                                 }
  $A2/>INT24ERRCODE/
             {mov [>Int24ErrCode],al              }
  $B0/$00/   {Mov al,0  ;renvoie IGNORE au programme}
  $89/$EC/   {mov sp,bp                           }
  $5D/       {pop bp                              }
  $CF);      {iret      ;Retour d'interruption    }

  end;{Int24}
```

```
Procedure SetInt24(Onoff:Boolean);
  {installe (onoff=true) ou désinstalle          }
  {(onoff=false) notre intr 24                   }
 Var
  regs:Record
   Ax,bx,cx,dx,bp,si,di,ds,es,flags:integer;
  end;
 begin
  Int24ErrCode:=0;
  With Regs Do
   If OnOff Then
   begin
    ax:=$3524;   {lecture de l'adr de      }
                 {l'int 24h                }
    Intr($21,Regs);
    If (OldInt24Ofs<>0) or (OldInt24Seg<>0) Then
    begin
     OldInt24Ofs:=Bx;
     OldInt24Seg:=Es;
    end;
    Dx:=Ofs(Int24);
    Ds:=Cseg;
    ax:=$2524;        {modif de l'adr de    }
                      {l'int 24h            }
    Intr($21,Regs);
   end
   else
    If (OldInt24Ofs<>0) or (OldInt24Seg <> 0) Then
    begin
     Ds:=OldInt24Seg;
     Dx:=OldInt24Ofs;
     ax:=$2524;
     Intr($21,Regs);
    end
end;{setint24}

begin {program-----------}

setint24(true); {installe notre int24h}

{Effectuons quelques tests avec la porte du    }
{drive A:> OUVERTE.                            }
```

```
{$I- }
 nom:='A:jerome';
 write('ASSIGN :');
 assign(fichier,nom);
 err:=Int24Result;
 If err<>0 then
  writeln('assign, Int24Err:',hi(err),
          ' TurboErr:', lo(err))
 else
  writeln('OK.......');


 write('RESET :');
 reset(fichier);
 err:=Int24Result;
 If err<>0 then
  writeln('reset, Int24Err:',hi(err),
          'TurboErr:', lo(err))
 else
  writeln('OK.......');

 write('REWRITE :');
 rewrite(fichier);
 err:=Int24Result;
 If err<>0 then
  writeln('rewrite, Int24Err:',hi(err),
          ' TurboErr:',lo(err))
 else
    writeln('OK.......');

  {restore l'int24 originale}
 setint24(false);

end.
```

On pourrait bien sûr adapter en fonction des erreurs. Par exemple, si le code d'erreur est 3, le programme peut inviter l'utilisateur à fermer la porte du lecteur.

6. Récapitulation

☞ **Il existe deux types de périphériques :**

- **Les périphériques bloc.**
- **Les périphériques caractères.**

☞ **Les périphériques peuvent être pilotés par des interruptions du BIOS. Depuis les versions MS-DOS 2.xx, les périphériques sont traités comme des fichiers logiques, ce qui permet l'utilisation des fonction 3Fh et 40h pour lire ou écrire sur un périphérique comme sur un fichier normal.**

☞ Le système des "handles" permet aussi de rediriger les entrées/sorties. Le périphérique standard de sortie d'un programme pourra ainsi devenir le périphérique standard d'entrée d'un autre programme. Cette technique est ce qu'on appelle un filtre.

☞ Il y a deux modes vidéo standard :

- Le mode monochrome qui divise l'écran en 25 lignes et 80 colonnes.

- Le mode CGA qui divise l'écran en mode graphique (200 lignes et 640 colonnes). Ce mode peut aussi émuler le mode texte.

☞ L'interruption de gestion du clavier renvoie toujours deux codes:

- Le code ascii.
- Le "scan-code".

☞ Tous les cas d'erreurs critiques sont gérés par l'interruption **24h**. Comme toutes les interruptions, l'interruption **24h** peut être réécrite pour s'adapter à des besoins précis.

Chapitre 4

Les répertoires

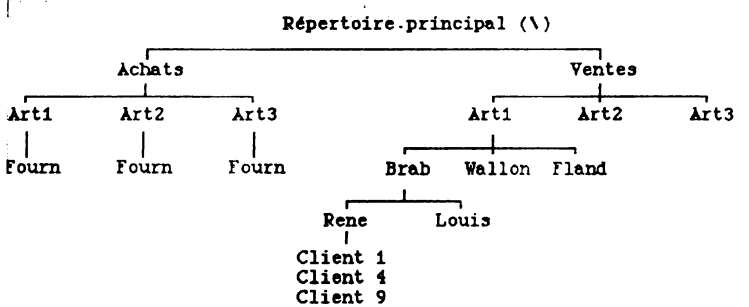
1. Généralités

L'un des principaux avantages des ordinateurs est leur capacité à manipuler des quantités plus ou moins importantes d'informations. Ces informations étant stockées dans des fichiers, le but premier d'un système d'exploitation est de fournir des services permettant de les organiser le plus efficacement possible.

Dans ce chapitre, nous allons passer en revue les techniques utilisées par MS-DOS au niveau de l'organisation des fichiers sur disque.

2. Les répertoires

Les informations sont donc stockées dans des fichiers résidant sur disque. Un disque pouvant contenir un nombre important de fichiers, les versions MS-DOS 2.xx et supérieures permettent de diviser le disque en un certain nombre de **répertoires** (directory) afin de grouper les fichiers par type. La structure des répertoires est une structure en "arbre" ou "hiérarchique": il y a un répertoire principal (Root directory), ce répertoire peut avoir un certain nombre de "fils" appelés sous-répertoires, ces sous-répertoires peuvent aussi avoir des fils, etc.



Chaque répertoire possède un nom. Le nom du répertoire principal est : (« \ »). L'utilisateur peut créer des nouveaux répertoires (en leur donnant à chacun un nom de son choix), il dispose aussi de commandes pour passer de l'un à l'autre, pour supprimer un répertoire, etc. Pour accéder à un fichier, un programme devra dorénavant spécifier son nom ainsi que le chemin ("path") permettant d'arriver au répertoire contenant le fichier.

Le **chemin** est la liste des noms des répertoires (séparés par le caractère « \ ») qu'il faut traverser pour arriver au répertoire contenant le fichier.

Cette référence au chemin peut éventuellement être précédée du nom du disque (cas où le répertoire ne se trouve pas sur le disque courant).

Lorsque COMMAND.COM (le processeur de commandes) possède le contrôle de l'ordinateur, c'est lui qui traite toutes les demandes de création de répertoire, changement, liste des fichiers, effacement, ... Pour cela, il utilise une série de fonctions du DOS qui sont aussi disponibles pour nos programmes, ces fonctions font toutes partie de l'interruption 21h, nous allons les analyser en détail.

Les fonctions de recherche de fichiers

existe deux fonctions de recherche des fichiers dans
n répertoire, elles sont incluses dans l'interruption
lh :

- Fonction **4Eh**, recherche du premier fichier.
- Fonction **4Fh**, recherche des fichiers suivants.

es fichiers appartenant à un répertoire font partie
'une liste, les éléments de cette liste peuvent être des
ichiers aussi bien que des noms de sous-répertoires.
lous disposons d'une fonction permettant de trouver
: nom du premier fichier (ou sous-répertoire) d'un
épertoire donné (**4Eh**), et d'une autre permettant de
:ouver les noms suivants (**4Fh**). On notera que lors
e la consultation du contenu d'un répertoire, il sera
ossible de ne demander que la liste des fichiers cor-
espondant à un "**masque**". Un masque permet, par
xemple, de ne demander que la liste de tous les fi-
hiers dont le nom commence par "PA" (PA*.*) ou
ncore la liste des fichiers dont l'extension est "ASM"
*.ASM), etc. Ce masque devra être spécifié lors de
appel aux fonctions de recherche du DOS.

La DTA - Disk Transfert Area

Les deux fonctions **4Eh** et **4Fh** renverrons les informations concernant les fichiers dans une zone mémoire appelée DTA.

La **DTA** est un tampon (buffer), une zone de la mémoire que MS-DOS utilisera pour stocker des données utilisées dans les opérations de lecture/écriture sur disque.

Cette zone peut se situer à n'importe quel endroit de la mémoire allouée à un programme et ce dernier pourra en spécifier l'adresse en utilisant la fonction **1Ah** de l'interruption **21h**. Il faut remarquer qu'il ne peut exister qu'une seule DTA et, si le programme ne spécifie rien, MS-DOS utilisera une zone par défaut d'une longueur de 128 bytes située à l'adresse **80h** du PSP. Le PSP est une zone de 256 bytes qui se trouve en mémoire avant un programme, nous y reviendrons en détail lors de l'étude de la mémoire (voir chapitre 7).

Note : La DTA était utilisée dans les versions 1.xx de MS-DOS pour les lectures/écritures sur fichier et son intérêt se restreint à quelques fonctions **(4Eh, 4Fh)** pour les versions MS-DOS **2.xx** et supérieures.

nsi, pour accéder à cette DTA, nous disposons de ois possibilités :

- Soit nous utilisons la DTA par défaut, elle se trouve à l'adresse **80h** du PSP; il nous faut alors connaître l'adresse de segment du PSP, mais la fonction **62h** de l'interruption **21h** (qui renvoie l'adresse du PSP) n'est possible qu'avec les versions MS-DOS **3.xx**.

- Soit nous utilisons la DTA par défaut et nous utilisons la fonction **2Fh** qui nous renvoie dans les registres ES et BX l'adresse de la DTA courante (version MS-DOS **2.xx** et supérieures).

- Soit nous installons notre propre DTA, il nous faudra alors sauver l'adresse de la DTA courante (fonction **2Fh** pour lire cette adresse) et puis nous utiliserons la fonction 1Ah pour spécifier l'adresse de notre DTA.

lors, quelle méthode choisir ? Si on souhaite comercialiser le programme, la première méthode n'est is à conseiller puisqu'elle empêche l'utilisation sur s ordinateurs qui n'ont pas le DOS 3.xx. La troisième éthode n'a pas beaucoup d'intérêt dans le cas préent, en effet on n'aura besoin de changer l'adresse de DTA que lorsqu'une zone de 128 bytes (taille de la TA courante) ne sera pas suffisante pour les ansferts de données, ce qui n'est pas le cas ici. La conde méthode semble donc la meilleure.

- La fonction **2Fh** lit l'adresse de la DTA courante.

- La fonction **1Ah** modifie l'adresse de la DTA courante.

- La fonction **62h** lit l'adresse du PSP qui contient la DTA par défaut.

L'interruption 21h, fonction 4Eh — Recherche du premier fichier.

Cette fonction permet de rechercher le premier fichier du répertoire spécifié. La recherche se fait en fonction du masque et seuls les fichiers correspondants aux attributs spécifiés sont sélectionnés.

Les registres suivants doivent être initialisés avant l'appel à la fonction :

AH	<-- **4Eh**
CX	<-- attributs de recherche
DS:DX	<-- adresse du masque de recherche

Le masque de recherche est une chaîne de caractères qui doit obligatoirement se terminer par un caractère dont le code ASCII est 0.

Le registre CX est un registre de 16 bits dont seulement certains seront utilisés :

Bit	Signification
0	Recherche des fichiers à lecture uniquement (read only).
1	Recherche des fichiers cachés.
2	Recherche des fichiers système.
3	Recherche du nom du volume (nom du disque).
5	Recherche des fichiers "archive".
6	Recherche des sous-répertoires.

Voici la signification des attributs :

- Les fichiers **Read Only** sont des fichiers pour lesquels la seule opération permise est la lecture, il est impossible d'écrire dans ces fichiers et on ne pourra pas les effacer.

- Les fichiers possédant l'attribut **Caché** ne seront pas listés par la commande DIR.

- Les fichiers **Système** sont ceux qui contiennent MS-DOS.

- L'attribut **Volume** indique qu'il s'agit du nom du disque, il ne peut y avoir qu'un seul fichier avec cet attribut.

- L'attribut **archive** signale les fichiers qui n'ont pas été modifiés depuis le dernier backup.

- L'attribut **sous-répertoire** indique qu'il s'agit du nom d'un sous-répertoire.

Si aucun de ces bits n'est à 1 (CX vaut donc 0), la recherche se fait sur les fichiers normaux; si un des bits ou une combinaison de ceux-ci est à 1, la recherche se fait sur les fichiers normaux et sur ceux spécifiés par CX.

ici quelques exemples:

CX		Type de recherche
CH	CL	
00000000	00000000	Fichiers normaux (en décimal, CX=0)
00000000	00000010	Fichiers normaux et cachés (CX=2)
00000000	00000100	Fichiers normaux et système (CX=4)
00000000	00010000	Fichiers normaux et sous-répertoires (CX=16)
00000000	00000110	Fichiers normaux et cachés et système (CX=2+4=6)

retour de la fonction **4Eh**, les informations suivantes sont renvoyées :

Si une erreur s'est produite, l'indicateur **Carry** (cf. registre FLAG) est différent de **0** et le registre AX contient le code d'erreur :

02h Indique "pas de fichier trouvé".
03h Indique que le répertoire spécifié n'existe pas.
12h Indique qu'il n'y a aucun fichier correspondant au masque spécifié.

S'il n'y a pas eu d'erreur, l'indicateur Carry = **0** et la DTA contient les informations suivantes :

Offset	Longueur	Signification
0	21	Réservé pour les appels à la fct 4Fh.

21	1	Attribut du fichier.
22	2	Heure du fichier (*).
24	2	Date du fichier (**).
26	2	Partie basse de la taille du fichier (***).
28	2	Partie haute de la taille du fichier (***).
30	13	Nom du fichier en format ASCIIZ sans espaces et avec un '.' séparant le nom et l'extension.

(*) Ces deux bytes de l'heure contiennent les informations suivantes :

Bits	Signification
0..4	Nombre de secondes à multiplier par 2.
5.10	Nombre de minutes (0..59).
11.15	Nombre d'heures (0..23).

Voici un exemple en Turbo Pascal illustrant le décodage de l'heure:

```
Procedure AfficheHeure(HeureX:Integer);

{1 Integer=2 Bytes}

Const

 {définition de masques "d'extraction"        }

 Secondes:INTEGER=$001E;   {0000 0000  0001 1111 }
                           {0    0     1    E    }
 Minutes:INTEGER=$07D0;    {0000 0111 1110  0000 }
                           {0    7    D    0    }
 Heures:INTEGER=$E800;     {1111 1000  0000 0000 }
                           {E    8    0    0  }
Var
 h,m,s:Integer;
```

```
begin

  h:=HeureX And Heures;
  {extrait les bits des heures}

  h:=h shr 11;
  {cadre les bits à droite dans h, SHIFT  }
  {RIGHT de 11 positions                  }

  m:=HeureX And Minutes;
  {extrait les bits des minutes}

  m:=m shr 5;
  {cadre les bits à droite dans m, SHIFT  }
  {RIGHT de 5 positions                   }

  s:=HeureX And Secondes;
  {extrait les bits des secondes          }
  {le cadrage des bits est correct comme  }
  {il s'agit des premiers bits            }

  s:=s*2;{il faut multiplier les sec. par 2}

end;
```

(**) Ces deux bytes de la date contiennent les informations suivantes:

Bits	Signification
0..4	Jour.
5..8	Mois (1..12).
9.15	Année à laquelle il faut ajouter 1980.

ATTENTION, la DTA ne doit pas être détruite car elle contient des informations qui seront utilisées par la fonction **4Fh** pour trouver les fichiers suivants dans le répertoire!

(***) La taille du fichier peut être calculée de la facon suivante:

```
Heure <-- Partie Haute*65536+Partie basse
```

Et en Pascal:

```
Procedure CalculeTaille(Haut,Bas:Integer);

var
 taille: REAL;

Begin

 Taille:=haut*65536.0 + Bas;

end;
```

L'interruption 21h, fonction 4Fh — Recherche des fichiers suivants.

Cette fonction doit obligatoirement être précédée d'un appel (1 seul) à la fonction **4Eh** pour trouver le **premier** fichier du répertoire. Elle pourra être appelée plusieurs fois pour trouver tous les fichiers, la recherche se termine quand la fonction renvoie une erreur. Seul le registre AH doit être initialisé avant l'appel à la fonction :

```
Ah <-- 4Fh
```

Aucun autre registre n'est nécessaire, cette fonction ira chercher les informations dont elle a besoin dans les 21 premiers bytes de la DTA qui a été utilisée par la fonction **4Eh, l'adresse de la DTA ne peut donc pas être modifiée entre les deux fonctions** !!!

Les informations renvoyées par cette fonction sont identiques à celles de la fonction **4Eh**.

Exemple de programmation

Voici un exemple (en Turbo Pascal) d'utilisation des deux fonctions (4Eh et 4Fh). A titre indicatif, nous avons précisé en commentaire (*...*) comment utiliser les méthodes 1 et 3. Ce programme utilise les fonctions suivantes de l'interruption 21h :

- **1Ah** Modifier l'adresse de la DTA courante.
- **2Fh** Lire l'adresse de la DTA courante.. -
- **30h** Lire la version du DOS.
- **36h** Lire le nombre de bytes disponibles sur un disque.
- **62h** Lire l'adresse du PSP dans BX.

```
Program Exemple;

Const
   {ces constantes représentent les valeurs }
   {des attributs à mettre dans CX: }

   Normal = 0;          { = 00000000 en binaire}
   Lecture = 1;         { = 00000001 en binaire}
   Cache = 2;           { = 00000010 en binaire}
   Systeme = 4;         { = 00000100 en binaire}
   Volume = 8;          { = 00001000 en binaire}
   Repertoire =16;      { = 00010000 en binaire}

Type

   Str64=String[64];  {la taille maximum d'un     }
                      {chemin ne peut excéder 64  }
                      {caractères                 }
```

```
    Registres = record  {définition des registres }
                        {du microprocesseur       }
    case Integer of
    1:(ax,bx,cx,dx,bp,si,di,ds,es,flags:Integer);
    2:(al,ah,bl,bh,cl,ch,dl,dh : Byte);
    end;

   (* définition de la DTA - Pour méthode 3 *)

    dtarec = RECORD
     PourDos : Array[1..21] OF Byte;
     Attr : Byte;
     DtaHeure,
     DtaDate,
     DtaTailleb,
     DtaTailleh : Integer;
     DtaNom:Array[1..13] OF Char;
    END;


Var
   reg : registres;     {registres            }
   methode: byte;       {indique la méthode   }
                        {utilisée (1,2 ou 3)  }

   DtaSeg,
   DtaOfs : Integer;    {seg, ofs de la DTA,}
                        {toutes les méthodes}

  (* Notre Dta - Méthode 3 *)
   Dta : DtaRec;


   SauveDtaSeg,
   SauveDtaOfs : Integer; {Seg, OFs de       }
                          {l'ancienne DTA    }
                          {Méth 3            }

Function GetPSP:Integer;

 (* Pour méthode 1 *)

 {Nous utilisons la DTA courante et nous       }
 {appelons donc MS-DOS à la rescousse pour     }
 {connaître l'adresse du PSP; il s'agit de la }
```

```
{fonction 62h disponible pour les versions    }
{MS-DOS 3.xx                                  }

Begin

 Reg.Ah:=$62;      {Ah <-- 62h, '$' est une}
                   {convention TURBO PASCAL}

 MsDos(reg);       {"Msdos" est une fonction de }
                   {Turbo Pascal pour appeler   }
                   {l'interruption 21h en lui   }
                   {transmettant les registres  }

 GetPsp:=Reg.bx;   {la fonction nous renvoie    }
                   {dans le reg BX l'adresse    }
                   {du segment du PSP:          }
                   {GetPsp <--Reg.BX            }

end;{fonction GetPSP}


Procedure Getdta;
  (* Méthode 2 et 3 *)

  {Lit et Sauve l'adresse de la DTA dans       }
  {DtaSeg et DtaOfs                            }

begin
   reg.ah := $2F;     {ah <-- 2Fh               }
   MsDos(reg);        {appel à MSDOS, interrupt }
                      {21h                      }
   DtaSeg:=reg.es;    {DtaSeg <-- ES = adr de   }
                      {Segment de la DTA        }
   DtaOfs:=reg.bx;    {DtaOfs <-- BX = offset   }
end;{Getdta}


Procedure SetDta(Segment,offset:integer);
  (* Méthode 3 *)

  {cette procédure est utilisée pour spécifier  }
  {l'adresse de notre DTA et restaurer l'adresse}
  {de la DTA courante à la fin du programme     }

Begin
   reg.ah := $1A;       {Ah <-- 1Ah              }
```

```
 reg.ds := Segment;  {DS <-- Segment de la DTA }
 reg.dx := Offset;   {DX <-- Offset de la DTA  }
 MsDos(reg);         {appel à MS-DOS           }
ND;{Setdta}


rocedure AfficheFichier;

{inspecte la DTA et affiche le nom du fichier}

ar
Attribut : byte;
Fichier : str64;
i : byte;

egin
{lecture de l'attribut, c'est le }
{22e byte de la DTA:             }
 Attribut := Mem[DtaSeg:DtaOfs+21];


 Attribut := Attribut And 22;
{22 = 00010110, pour ne tenir compte que }
{des bits 1,2,4)                         }

{lecture du nom du fichier à partir de       }
{l'offset 30 de la Dta, et on cherche jusqu'à}
{ce que le caractère lu = ASCII              }
{                                            }
{note: Si ord(caract) = code ASCII du caract,}
{      alors Chr(codeASCII) = caract         }


 Fichier:='';
 i:=0;

 While Ord(Mem[DtaSeg:DtaOfs+30+i]) <>0 Do
 Begin
  Fichier:=Fichier+
  Chr(ord(Mem[DtaSeg:DtaOfs+ 30+i]));

  i:=i+1;
 end;

[on s'arrange pour que les fichiers soient   }
[sur 13 caractères                           }
```

```
  While (length(fichier)<13) Do
  fichier:=Fichier+' ';

 {on ajoute quelques blancs...}
  Fichier:=Fichier+' ';

 {on ajoute l'attribut}
  Case Attribut OF
   Normal :     Fichier:=Fichier+'Normal';
   Cache :      Fichier:=Fichier+'Caché';
   Systeme :    Fichier:=Fichier+'Système';
   Repertoire:  Fichier:=Fichier+'<DIR>';
  else
   Fichier:=Fichier+'Attribut Inconnu';

  end;{Case attribut}


 {on affiche le fichier et son attribut...     }
 {on pourrait aussi ajouter l'heure et la      }
 {date avec les proc. ci-dessus :              }

  writeln(fichier);

end;{affiche fichier}


Function
BytesDisponibles(disque:byte):real;

 {retourne le nombre de bytes disponibles,     }
 {disque   = 0 pour le disque courant          }
 {         = 1 pour le disque 'A:'             }
 {         = 2 pour le disque 'B:'             }
 {         = ...                               }
 {dans ce programme, disque=0                  }

Begin

  reg.ah:=$36;       {ah <-- 36h, lecture de    }
                     {l'espace disque libre     }
  reg.dl:=disque;    {dl <-- Numéro du disque   }
  msdos(reg);        {appel à MS-DOS            }

 {Cette fonction retourne: (cf. chap. 3.D.2)    }
 { ax = Nombre de secteur/cluster               }
```

```
  { bx = nombre de clusters disponibles      }
  { cx = nombre de bytes par secteur         }
  { dx = nombre de clusters sur le disque    }

   If reg.ax=$ffff

   Then
   {indique que le num. du disque était       }
   {incorrect                                 }
   Begin
    bytesDisponibles:=0.0;
    writeln('Spécification du disque
            incorrecte');
   end
   else
    bytesDisponibles:=1.0*reg.bx*reg.ax*reg.cx;
end; {BytesDisponibles}


Procedure
Directory(Masque:Str64;Attribut:Integer);

{Nous cherchons dans le répertoire courant    }
{tous les fichiers correspondant au masque    }
{spécifié, pour cela nous utilisons d'abord la}
{fonction 4Eh de l'interruption 21h pour      }
{trouver le premier fichier de la liste.      }
{Ensuite nous appelons la 4Fh jusqu'à obtenir }
{une erreur, ce qui nous indique qu'il n'y a  }
{plus de fichiers correspondant au masque     }
{dans le répertoire                           }

Var

  ok:Boolean;   {variable = True Si le flag   }
                {Carry=0 et =false si le flag }
                {est <> 0                     }

Begin
  Masque:=Masque+Chr(0);

 {le masque doit être en format ASCIIZ, ce    }
 {qui signifie qu'il doit se terminer par le  }
 {caractère = ASCII 0                         }

  Reg.Ah:=$4E;              {$4E= convention   }
```

```
                                {pascal nombre hexa }
 Reg.CX:=Attribut;              {CX contient        }
                                {l'attribut         }
 Reg.DS:=Seg(Masque[1]);        {DS = Adr de        }
                                {segment du masque  }
 Reg.DX:=Ofs(Masque[1]);        {DX = Offset de     }
                                {masque             }

{On prend la position 1 de masque car en     }
{Turbo Pascal, la position 0 contient la     }
{longueur du string                          }

 MsDos(reg);
 Ok:= (Reg.flags And $0001)=0; {ok= True Si }
                               {le flag     }
                               {carry = 0   }

 If Not(Ok)

 Then
 Begin
  Writeln(^G,'Aucun fichier pour le masque
              spécifié...');
  delay(500);
  exit;
 end
 Else
  AfficheFichier;

 Repeat
  Reg.Ah:=$4F;    {il ne faut plus spécifier }
                  {de masque, la DTA         }
  MsDos(reg);     {contient les infos        }
                  {spécifiées lors de        }
                  {l'appel à la fonction 4Eh }

  Ok:= (Reg.flags And $0001)= 0 ;
 {ok=True Si le flag carry=0}

  If Ok then
   AfficheFichier
 Until Not(ok);{teste le flag "carry"}

{on peut aussi afficher le nombre de bytes   }
{disponibles                                 }
 writeln;
```

```
    writeln(bytesDisponibles(0):1:0,' octets
    disponibles...')
  end;{Procedure Directory}


Function VersionDos:byte;
   (* méthode 1 *)

  {lit la version du DOS car la fonction 62h    }
  {utilisée n'est disponible que pour les       }
  {versions MS-DOS 3.xx                         }

Begin
   Reg.ah := $30;    {Ah <--fonction 30h, lit   }
                     {la version du DOS         }
   MsDos(reg);       {appel à MS-DOS au retour: }
                     {AL = partie entière de la }
                     {version                   }
                     {AH = partie fractionnaire }

   VersionDos:=Reg.al;    {seule la partie      }
                          {entière nous         }
                          {intéresse            }
end;{VersionDos}


Begin{----program-------------------------}

   Write ('Quelle méthode ? (1,2,3)');
   Read(methode);
   clrscr;{efface l'écran}

   If not(methode in [1..3])
   Then     {méthode <> 1,2 ou 3}
   Begin    {oui ==> erreur}
    writeln(^G,'Méthode incorrecte...');
    halt;
   end;

   If methode=1 Then
    If VersionDos<3
    Then    {DOS 1.xx ou 2.xx ? }
    Begin   {oui ==> erreur}
     writeln(^G,'La Méthode 1 nécéssite un DOS
                 3.xx...');
     halt;
```

```
  end;

Case methode Of  {init le seg et l'offset   }
                 {de la dta en fonction de  }
                 {la méthode                }
  1:Begin
      DtaSeg:=GetPsp;
      DtaOfs:=$80;     {80h = offset de Dta  }
                       {courante dans PSP    }
    end;

  2:Begin
      GetDta;          {cette proc init DtaSeg }
                       {et DtaOFs              }
    end;

  3:begin
      GetDta;          {cette proc init        }
                       {DtaSeg et DtaOFs       }
      SauveDtaSeg:=DtaSeg;  {il faut sauver    }
                            {ces valeurs       }
      SauveDtaOfs:=DtaOfs;
      Dtaseg:=seg(dta);     {dtaseg <-- seg    }
                            {de notre DTA      }
      DtaOfs:=Ofs(Dta);     {dtaofs <-- ofs    }
                            {de notre DTA      }
      SetDta(DtaSeg,DtaOfs);{installe notre    }
                            {Dta               }
  end;

end; {case}

writeln('Masque = *.*, Attribut normal');
writeln('==============================');
Directory('*.*',Normal);
Writeln('tapez sur une touche pour
continuer');
Repeat until Keypressed;

writeln('Masque = *.*, Attribut normal et
sous-répertoire');
writeln('========================');
Directory('*.*',Normal+Repertoire);
Writeln('tapez sur une touche pour
continuer');
Repeat until Keypressed;
```

```
  writeln('Masque = \DOS\PA??????.*, Attribut
  normal, caché');
  writeln('=========================');
  Directory('\DOS\PA??????.*',Normal+Cache);
  Writeln('tapez sur une touche pour
  continuer');
  Repeat until Keypressed;

 {...}

  If methode=3 then
   SetDta(SauveDtaSeg,SauveDtaOfs);  {restore
                                      Dta
                                      originale}

end.        {program}
```

Comment faire s'il y a une référence à un disque dans le masque spécifié ? Pour les deux fonctions de recherche, cela ne pose aucun problème, tout est géré par MS-DOS; par contre la fonction qui renvoie le nombre de bytes disponibles ne fonctionnera plus... Pour résoudre ce problème, il faudra tester s'il y a une référence à un disque dans le masque et convertir la lettre en un numéro (0 si pas de référence à un disque) :

```
...
 var

  i:byte;
  lettre:string[1];

 Begin

  i:=Pos(':',Masque);  {regarde si il y a       }
                       {un ':' dans le masque   }
  If i=0 then
   disque:=0           {non, alors le num. du   }
                       {disque= 0               }
```

```
  Else
   begin
    lettre:=Copy(masque,i-1,1);
    {lettre <-- copie de masque à partir de la
    position i-1 sur 1 caractère}

    disque:=ord(lettre[1])-65;
    {disque <-- Code ASCII de la lettre- 64 (65 est
    le code ASCCI de 'A'}
   end;
...
```

4. Les fonctions de gestion des répertoires

Il existe 4 fonctions de gestion des répertoires, elles font toutes partie de l'interruption 21h :

Fonction **39h**	-	Création d'un répertoire.
Fonction **3Ah**	-	Effacement d'un répertoire.
Fonction **3Bh**	-	Changement de répertoire.
Fonction **47h**	-	Lecture du répertoire courant.

A part le fait qu'on ne peut évidemment pas effacer le répertoire courant, ces fonctions ne présentent aucune particularité.

5. Autres fonctions

Parmi les autres fonctions concernant les fichiers, on notera :

Fonction **57h**	-	Lecture/Ecriture de la date/ heure.
Fonction **43h**	-	Lecture/Ecriture des attributs d'un fichier.

6. Récapitulation

☞ Les disques sont divisés en répertoires pour permettre de classer les fichiers par type.

☞ Le **chemin** est la liste des noms des répertoires qu'il faut traverser pour arriver au répertoire contenant le fichier.

☞ Les fichiers d'un répertoire font partie d'une liste. La fonction **4Eh** permet de rechercher le premier fichier d'un répertoire et la fonction **4Eh** permet de rechercher les fichiers suivants. Ces fonctions sont incluses dans l'interruption **21h**.

☞ La **DTA** est une zone mémoire utilisée par MS-DOS lors de certaines opérations sur disque.

Chapitre 5

Manipulation de fichiers

1. Les fonctions du DOS

Un programme pourra lire ou écrire des informations dans une structure appelée **"fichier"**. Les fichiers sont identifiés par un nom et on distingue les fichiers **logiques** et **physiques**. Le clavier, l'imprimante, l'écran sont des ***fichiers logiques***; les autres résident sur disque et sont destinés à stocker les informations de manière durable, ce sont les ***fichiers physiques***. Les fichiers stockés sur disque pourront être regroupés dans des répertoires.

Les fonctions de base de manipulation des fichiers sont les suivantes, elles font toutes partie de l'interruption **21h** :

3Ch	Crée un fichier (écrase si existe déjà).
5Bh	Crée un fichier (erreur si existe déjà).
3Dh	Ouvre un fichier existant.
3Eh	Ferme un fichier.
3Fh	Lecture.
40h	Ecriture.
41h	Efface un fichier.
42h	Ajuste le pointeur de fichier.
56h	Renomme un fichier.
5Ch	Protège/déprotège une région d'un fichier (LAN).

2. Les enregistrements

Un fichier n'est en fait qu'un ensemble de bytes stockés sur disque. Comme il est souvent destiné à contenir un nombre plus ou moins important d'informations d'un même type (fichier de clients, d'articles,...), il sera alors divisé logiquement en **enregistrements** (records); chaque enregistrement contiendra une des informations. La taille et le type des enregistrements est déterminée ***lors de l'utilisation du fichier***, le nombre d'enregistrements n'est limité que par l'espace disque disponible.

Note : Certains langages de programmation (comme TURBO PASCAL) limitent le nombre d'enregistrements d'un fichier à 65536. Cette limite est due au fait que des nombres de type INTEGER sont utilisés pour spécifier le numéro des enregistrements et que l'intervalle des valeurs de ce type est limité à 65536; ce n'est donc pas une limite du MS-DOS !

La méthode la plus courante d'accès à un enregistrement est l'utilisation d'un numéro; ce numéro servira à calculer un offset (du premier byte de l'enregistrement) par rapport au début du fichier :

offset = Numéro*nombre de bytes par enregistrement.

Le premier enregistrement d'un fichier aura toujours le numéro 0.

3. Les "handles"

A partir des versions 2.xx de MS-DOS, les fichiers et les périphériques sont gérés par des "handles". L'idée consiste à traiter de la même manière les données stockées dans un fichier "physique" résidant sur disque et les données à écrire ou lire sur un périphérique appelé fichier "logique".

> Un "**handle**" est un numéro qui est attribué à un fichier lorsqu'on l'utilise.

Un programme ne pourra en théorie pas travailler avec plus de 20 handles ***en même temps***. Les "handles" numérotés de 0 à 4 sont d'office attribués aux périphériques standard :

0 Clavier.
1 Ecran.
2 Ecran.
3 Port série.
4 Imprimante.

Les autres handles (5 à 19) sont libres pour les fichiers. Lorsqu'on souhaitera travailler avec un fichier, il faudra l'**ouvrir** : MS-DOS attribuera alors un handle libre à ce fichier et nous devrons spécifier ce handle lors de chaque opération sur le fichier. Dès que nous n'aurons plus besoin du fichier, nous le fermerons : MS-DOS récupérera alors le numéro du

handle pour l'attribuer lors d'une prochaine demande d'ouverture de fichier.

Les opérations d'ouverture, de fermeture, de lecture, d'écriture sur des fichiers peuvent être réalisées grâce à certaines fonctions de l'interruption **21h**. La méthode d'accès à un fichier existant est la suivante :

1. Il faut ouvrir le fichier : la fonction **3Dh** renvoie un "handle" qui devra être spécifié lors de toutes les opérations sur ce fichier1.

2. Il faudra ensuite utiliser la fonction **42h** pour spécifier le numéro de l'enregistrement qui sera manipulé (positionnement du "pointeur" de fichier).

3. On pourra maintenant appeler les fonctions **3Fh** ou **40h** pour lire ou écrire un enregistrement (celui indiqué par le pointeur de fichier).

4. Les points 2 et 3 pourront être répétés.

5. Finalement, le fichier devra être fermé (fonction **3Eh**) pour libérer le handle (qui pourra être attribué à un autre fichier). Lors de cette opération, MS-DOS ajustera la taille du fichier dans le répertoire. Si cela n'est pas fait, les enregistrements ne pourront pas être consultés la prochaine fois : MS-DOS considérera qu'on essaie de lire après la fin du fichier !

Notes :

- **Il est intéressant d'utiliser ces fonctions car elles sont prévues pour travailler en réseau local et bien qu'étant prévues pour le réseau MS-NET (Microsoft), toutes les fonctions de base (citées ci-dessus) sont notamment compatibles avec le système NOVELL. On peut donc être assuré qu'un programme utilisant ces fonctions sera compatible avec la plupart des réseaux du marché !!!**

- Dans le cas d'un langage comme Turbo Pascal, les fonctions internes de gestion des fichiers se "plantent" en cas d'accès simultané en réseau local.

Nous allons maintenant développer un ensemble de procédures qui pourront être utilisées dans tous nos programmes en Pascal. Les variables communes à toutes les procédures sont les suivantes:

```
Const
 DosIntr = $21;{ Numéro de l'interrupt. DOS}

Type

 Chemin = string[67];

 Registres = RECORD
  CASE Integer OF
   1:(ax,bx,cx,dx,bp,si,di,ds,es,flags:Integer);
   2:(al,ah,bl,bh,cl,ch,dl,dh : Byte);
  END;

var
 Regs : Registres;
 ErrCode : Integer;
```

Les deux procédures suivantes seront aussi nécessaires:

```
Procedure
RealEnDoubleMot(r:Real;
                Var HW,LW:Integer);

  {Cette procédure convertit un Réel en une    }
  {représentation sur 32 bits (double mot)     }

Var
  Temp :Real;
begin
   HW := trunc(R/65536.0);
   Temp:= R - (HW * 65536.0);
   if (Temp > 32767.0) then
   begin
    Temp := Temp - 32767.0;
    LW := trunc(Temp) + 32767
   end
   else
    LW := trunc(Temp);
end;{RealEnDoubleMot}

Procedure
DoubleMotEnReal(HW,LW:Integer;
                Var R:Real);

{Cette procédure convertit un nombre sur 32  }
{bits en un nombre réel.                     }

Var
 temp:Real;

begin
   Temp:=LW;
   If temp<0.0 then temp:=temp+65536.0;
   R:=HW*65536.0+Temp;
end;{DoubleMotEnReal}
```

Nous conseillons de regrouper les déclarations de variables, les deux procédures ci-dessus et celles qui suivent dans l'ordre d'apparition dans un même fichier FICH.PAS qui sera inclu dans chaque programme grâce à la directive : {$I FICH.PAS}.

La fonction 59h — Détermination d'erreur.

Toutes les fonctions de gestion de fichiers renvoient le flag carry = 1, si une erreur s'est produite, et le code de l'erreur est dans le registre AX. Depuis les versions 3.xx de MS-DOS, la fonction 59h permet d'obtenir des informations complémentaires :

- Le code de l'erreur.
- La classe de l'erreur.
- L'action recommandée.
- L'origine de l'erreur.

Cette fonction pourra donc être appelée dans tous les cas pour déterminer s'il y a eu erreur et agir éventuellement en conséquence.

```
Procedure Erreur(ApresFin : integer);

{Cette procédure est appelée après chaque      }
{opération sur les fichiers pour obtenir       }
{plus d'informations en cas d'erreur, elle     }
{appelle la fonction 59h                       }

type
 str80=String[80];

var
 Pause : char;
 ErrMsg : str80;
 ClsMsg : str80;
 ActMsg : str80;
 LocMsg : str80;

 ClsCode: Integer;
 ActCode: Integer;
```

```
 LocCode: Integer;

begin

 if ApresFin <> 0 then
 begin
  writeln;
  writeln ('Fin de fichier atteinte');
  writeln (ApresFin,' bytes n''ont pas été lus
           ou écrits.');
 end;

 if ErrCode = 0 then
 exit;

 with Regs do
 begin
  Ax := $5900; {Fonction 59h}
  Bx := 0;
  Intr(DosIntr,Regs);
  if (Ax <> 0) then
  begin
   ClsCode := Hi(Bx);
   ActCode := Lo(Bx);
   LocCode := Hi(Cx)
  end;
  ErrCode := Ax;
end;{with}

Case ErrCode of
 0 : ErrMsg := 'Pas erreur';
 1 : ErrMsg := 'Numéro de fonction
                incorrect';
 2 : ErrMsg := 'Fichier pas trouvé';
 3 : ErrMsg := 'Chemin pas trouvé';

 {ajouter ici les autres messages, cf.   }
 {fonction 59h en fin d'ouvrage, codes   }
 {d'erreur                               }

else
 ErrMsg:='Consulter les codes en fin de
          livre...';
end;
```

```
 Case ClsCode of
  1 : ClsMsg := 'Plus de ressource';
  2 : ClsMsg := 'Situation temporaire';

 {ajouter ici les autres messages, cf.    }
 {fonction 59h en fin d'ouvrage, classe   }
 {d'erreur                                }

 else
  ClsMsg:='Consulter les codes en fin
           livre...';
 end;

 Case ActCode of
  1:ActMsg:='...';

 {ajouter ici les autres messages, cf.     }
 {fonction 59h en fin d'ouvrage, action à  }
 {prendre                                  }
 {......                                   }

 else
  ActMsg:='Consulter les codes en fin de
           livre...';
 end;

 Case LocCode of
  1 : LocMsg := 'Inconnu';
  2 : LocMsg := 'Disque';
  3 : LocMsg := 'Réseau';
  4 : LocMsg := 'Port Série';
  5 : LocMsg := 'Mémoire';
 end;

 writeln;
 writeln;
 writeln(ErrMsg);
 writeln(ClsMsg);
 writeln(ActMsg);
 writeln(LocMsg);
 writeln;
 write('Tapez sur une touche pour continuer');
 read(KBD,Pause);
 writeln;
 end; {Procédure Erreur }
```

La fonction 42h — Positionnement du pointeur de fichier.

Chaque opération de lecture ou d'écriture dans un fichier devra être précédée du positionnement du pointeur de fichier. Comme les enregistrements sont atteints au moyen d'un numéro, cette procédure aura les paramètres d'entrée suivants : le handle du fichier, le numéro de l'enregistrement et la taille de l'enregistrement. Les deux dernières informations permettront à la procédure de calculer un déplacement en bytes par rapport au début du fichier.

```
Procedure
Positionne(Handle : integer;
           RecNum : real;
           Taille : integer);

{Appelle la fonction 42h pour positionner     }
{le pointeur de fichier sur l'enregistrement  }
{RECNUM du fichier                            }
{Cette procédure est appelée avant toute      }
{opération de lecture ou d'écriture           }

var
 HighWord : integer;       {Mot le plus  }
                           {significatif }
 LowWord : integer;        {Mot le moins }
                           {significatif }

 Position : real;    {Offset du premier Byte }
                     {de l'enreg.            }

begin
```

```
  Position := RecNum*Taille;
  RealEnDoubleMot(Position,HighWord,LowWord);

  with Regs do
  begin
   Ax := $4200;      {Fonction 42h,      }
                     {Méthode 0          }
   Bx := Handle;
   Cx := HighWord;
   Dx := LowWord;
   Intr(DosIntr,Regs);
   if ((Flags and 1) <> 0) then
   ErrCode := Ax
  else
   ErrCode := 0;
  end;
  Erreur(0);

end;{ Positionne }
```

Cette même fonction **42h** peut aussi être utilisée pour calculer la taille du fichier. Le principe consiste à demander au DOS de positionner le pointeur de fichier à 0 bytes de la fin du fichier, la valeur du pointeur exprimée en bytes représentera alors la taille du fichier.

```
function
TailleFichier(Handle : integer) : real;

{Cette fonction renvoie la taille du          }
{fichier exprimée en bytes; elle utilise      }
{pour cela la fonction 42h en lui demandant   }
{de positionner le pointeur à 0 bytes de la   }
{fin du fichier, la fonction renvoie un       }
{offset de cette position par rapport au      }
{début du fichier ==> c'est le nombre de      }
{bytes du fichier                             }


Var
 temp :real;
```

```
begin
  with Regs do
   begin
    Ax := $4202;          {Fonction 42h,          }
                          {Méthode 2              }
    Bx := Handle;
    Cx := 0;              {Offset 0 à partir de   }
                          {la fin                 }
    Dx := 0;
    Intr(DosIntr,Regs);
    if ((Flags and 1) <> 0) then
    begin
     ErrCode := Ax;
     TailleFichier := -1;
    end
    else
    begin
     DoubleMotEnReal(Dx,Ax,Temp);
     TailleFichier:=Temp;
     ErrCode := 0
    end;
   end;{with}
   Erreur(0);
end; { TailleFichier }
```

La fonction 3Eh — Fermeture de fichier.

Cette fonction libère le handle attribué à un fichier dont on n'a plus besoin. Elle a aussi un effet important qui consiste à mettre à jour la taille actuelle du fichier dans le répertoire. Si cette taille n'est pas correcte, toute tentative ultérieure pour lire les bytes situés "après" la fin théorique (déterminée par la taille dans le répertoire) du fichier provoquera une erreur!

```
Function Ferme(Handle : integer):integer;

 {Cette fonction ferme le fichier          }
 {correspondant au handle donné            }
begin
  with Regs do
  begin
    Ax := $3E00;           {interruption 21h, }
                           {fonction 3Eh      }
    Bx := Handle;          {Bx <-- handle     }
    Intr(DosIntr,Regs);
    if ((Flags and 1) <> 0) then
    {teste flag CARRY  }
     ErrCode := Ax
    else
     ErrCode := 0
  end;
  Ferme:=ErrCode;
  Erreur(0);
end; { Ferme }
```

La fonction 3Dh — Ouverture d'un fichier.

Cette fonction doit être appelée une seule fois avant toute opération sur un fichier. Elle renverra alors un "handle" (un numéro) qui devra être spécifié comme paramètre de TOUTES les fonctions manipulant ce fichier. Le numéro est unique, il ne pourra être attribué à un autre fichier qu'après la fermeture du fichier auquel il est actuellement assigné. Si le fichier n'existe pas, la fonction renvoie -1 comme handle pour signaler l'erreur.

```
function Ouvre(NomFich:Chemin):integer;

{cette fonction ouvre un fichier et       }
{renvoie le handle attribué par MS-DOS    }

var
 NomfichZ:Chemin;     {Nom du fich en format  }
                      {ASCIIZ                 }
 Lg:integer;

begin

  lg := length(NomFich);
  move(NomFich[1],NomfichZ,Lg);
  NomfichZ[Lg] :=#0;      {Doit être en   }
                          {format ASCIIZ  }

  with Regs do
  begin
   Ax := $3D42;     {Fonction 3Dh, accès     }
                    {lecture/écriture        }
   Ds := Seg(NomfichZ);
   Dx := Ofs(NomfichZ);
   Intr(DosIntr,Regs);
```

```
    if ((Flags and 1) <> 0) then
    begin
     ErrCode := Ax;
     Ouvre := -1;
    end
    else
    begin
     ErrCode := 0;
     Ouvre := Ax;
    end;
  end;{with}
  Erreur(0);
end;{ Ouvre }
```

La fonction 3Ch — Création de fichier.

Cette fonction crée un fichier dont le nom est spécifié comme paramètre. Si le fichier existe déjà, il est écrasé. Le fichier est aussi ouvert par cette opération.

```
Function Cree(NomFich : Chemin):Integer;

{Cette fonction crée un fichier          }
{(écrase si existe) et l'ouvre           }
{Elle renvoie le handle attribué         }
{par MS-DOS                              }
var
 NomfichZ : Chemin;     {Nom du fich.     }
                        {en format ASCIIZ }
 Lg : integer;
 Handle : integer;

begin

  Lg := length(NomFich);
  move(NomFich[1],NomfichZ,Lg);
  NomfichZ[Lg] := #0;    {Doit être en   }
                         {format ASCIIZ  }

  with Regs do
  begin
   Ax := $3C00;      {fonction 3Ch, écrase   }
                     {si existe              }
   Ds := Seg(NomfichZ);
   Dx := Ofs(NomfichZ);
   Cx := 0;
   Intr(DosIntr,Regs);
   if ((Flags and 1) <> 0) then
   begin
    ErrCode := Ax;
    cree:=-1;
   end
   else
```

```
    begin
     ErrCode := 0;
     Handle := Ax;
     cree:=handle;
    end
   end;
   Erreur(0);
end;{ Cree }
```

La fonction 41h — Effacement de fichier.

Cette procédure efface le fichier dont le nom est spécifié comme paramètre d'entrée. Le fichier doit être fermé.

```
procedure Efface(NomFich : Chemin);

{Cette fonction efface un fichier}
var
 NomfichZ : Chemin;       {Nom du fich. en      }
                          {format ASCIIZ        }
 Lg : integer;

begin

  Lg := length(NomFich);
  move(NomFich[1],NomfichZ,Lg);
  NomfichZ[Lg] := #0;

  with Regs do
  begin
   Ax := $4100; {fonction 41h}
   Ds := Seg(NomfichZ);
   Dx := Ofs(NomfichZ);
   Intr(DosIntr,Regs);
   if ((Flags and 1) <> 0) then
    ErrCode := Ax
   else
    ErrCode := 0;
  end;{with}
  Erreur(0);
end;{ Efface }
```

La fonction 3Fh — Lecture d'un enregistrement.

Cette procédure lit un enregistrement dans le fichier associé au handle spécifié. Le numéro d'enregistrement et la taille de celui-ci font partie des paramètres d'entrée. Elle appelle la procédure de positionnement du pointeur de fichier définie ci-dessus. L'enregistrement est lu dans la variable de sortie "buffer".

```
procedure LitEnreg(Handle : integer;
                   RecNum : real;
                   Taille : integer;
                   var Buffer);

{Cette fonction lit l'enregistrement RECNUM  }
{du fichier, elle appelle POSITIONNE pour    }
{ajuster le pointeur de fichier avant        }
{l'opération. Les données lues sont          }
{transférées dans BUFFER                     }


var
 BytesLus : integer;    {Nombre de       }
                        {bytes lus       }
 ApresFin : integer;    {Nombre de       }
                        {bytes pas lus   }
begin

  Positionne(Handle, RecNum, Taille);

  with Regs do
  begin
   Ax := $3F00; {fonction 3Fh}
   Bx := Handle;
   Ds := Seg(Buffer);
   Dx := Ofs(Buffer);
   Cx := Taille;
```

```
    Intr(DosIntr,Regs);
    if ((Flags and 1) <> 0) then
    begin
     BytesLus := 0;
     ErrCode := Ax
    end
    else
    begin
     BytesLus := Ax;
     ErrCode := 0
    end;
  end;{with}
  ApresFin := Taille - BytesLus;
  Erreur(ApresFin);
end; { LitEnreg }
```

La fonction 40h —Ecriture d'un enregistrement.

Cette procédure écrit un enregistrement dans le fichier dont le handle est spécifié comme paramètre d'entrée. Comme "LireEnreg", elle appelle la procédure de positionnement du pointeur de fichier définie ci-dessus.

```
procedure ModifEnreg(Handle : integer;
                     RecNum : real;
                     Taille : integer;
                     VAR Buffer);

{Cette fonction Ecrit le contenu de BUFFER  }
{dans l'enregistrement RECNUM du fichier,   }
{elle appelle POSITIONNE pour ajuster le    }
{pointeur de fichier avant l'opération      }


var
 Ecrits : integer;        {Nombre de bytes   }
                          {écrits            }
 ApresFin : integer;      {Nombre de bytes   }
                          {pas écrits        }

begin
  Positionne(Handle, RecNum, Taille);
  with Regs do
  begin
   Ax := $4000;           {fonction 40h      }
   Bx := Handle;          {Bx<--Handle       }
   Cx := Taille;          {Cx<--Nombre de    }
                          {bytes à écrire    }
   Ds := Seg(Buffer);{}
   Dx := Ofs(Buffer);{}
   Intr(DosIntr,Regs);
   if ((Flags and 1) <> 0) then
   begin
```

```
    Ecrits := 0;
    ErrCode := Ax
   end
   else
   begin
    Ecrits := Ax;
    ErrCode := 0
   end;
  end;{with}
  ApresFin := Taille - Ecrits;
  Erreur(ApresFin);
end; { ModifEnreg }
```

Pour ajouter un nouvel enregistrement au fichier, on calcule d'abord le nombre d'enregistrements du fichier. Comme les enregistrements sont numérotés à partir de 0, ce nombre correspond au numéro du dernier enregistrement+1. On appelle ensuite la procédure "ModifEnreg" que nous venons de définir :

```
procedure AjouteEnreg(Handle : integer;
                      Var RecNum : real;
                      taille : integer;
                      var Buffer);

{Cette fonction ajoute un nouvel              }
{enregistrement au fichier. Elle calcule      }
{le numéro d'enregistrement qui = le nombre   }
{d'enregistrements du fichier                 }


begin
  RecNum :=
   trunc((TailleFichier(Handle))/Taille);

  ModifEnreg(Handle,recnum,Taille,Buffer);
end; { AjouteEnreg }
```

La fonction 45h — Mise à jour de la taille du fichier dans le répertoire.

Idéalement, pour éviter qu'en cas de panne de courant la taille du fichier dans le répertoire soit incorrecte, cette procédure devrait être appelée après chaque ajout d'un nouvel enregistrement dans le fichier. Cette opération de mise à jour prend un peu de temps mais elle est beaucoup plus rapide que si l'on devait fermer et réouvrir le fichier.

```
procedure MajDir(Handle : integer);

{Cette procédure utilise la fonction 45h    }
{pour obtenir un second handle pour le      }
{fichier, et elle ferme le second handle,   }
{cela a pour effet de mettre à jour la      }
{taille du fichier sur disque et évite de   }
{devoir réouvrir le fichier puisqu'on a     }
{toujours le premier handle                 }


begin
 with Regs do
 begin
  Ax := $4500;  {fonction 45h}
  Bx := Handle;
  Intr(DosIntr,Regs);
  if ((Flags and 1) <> 0) then
   ErrCode := Ax
  else
  begin
   errCode:=Ferme(Ax);
  end;
 end; {with}
 Erreur(0);
end; { MajDir }
```

Exemple de programmation

Nous allons maintenant réaliser un petit programme qui illustre l'utilisation de ces nouvelles procédures. Ce programme peut :

- Créer un fichier.
- Ouvrir un fichier.
- Fermer un fichier.
- Ajouter un enregistrement.
- Chercher et lire un enregistrement.
- Modifier un enregistrement.
- Lister des enregistrements.

```
Program Fichier;

{$I FICH.PAS}       {Fichier contenant          }
                    {les nouvelles procédures   }
                    {définies ci-dessus         }
type
 Str12=string[12];
 Str64=string[64];

{Structure pour stocker les   }
{informations d'un fichier    }
 FichRec=Record
    Nom : string[64];  {Nom du fichier   }
    Hdl : Integer;     {handle           }
    RnActuel:integer;
 end;

 {Structure d'un enregistrement   }
 ArticleRec=Record
    Code :str12;
    designation :str64;
    prix :Real;
 end;
```

```
 Var
  ArtEnreg:ArticleRec;
  article :FichRec;

  chh,menu:char;
  ok :boolean;
  err :Integer;

 Procedure message(col,lin,Num:Integer;
                   delai :Boolean);

  {Affiche un message Numéro "num" à     }
  {la colonne "col" et ligne "lin"       }

begin

  gotoxy(col,lin);
  case num of
   0:Clreol;
   1:write(^G,'Erreur, le fichier ARTICLE est
             déjà ouvert');
   2:write(^G,'Erreur, le fichier ARTICLE
             n''est pas ouvert');
   3:write(^G,'Impossible de créer le
             fichier');
   4:write(^G,'Fichier inexistant');
   5:write(^G,'Fichier créé et ouvert...,
             handle=');
   6:write(^G,'Fichier ouvert..., handle=');
   7:write(^G,'Fichier fermé...');
   8:write(^G,'ATTENTION, toutes les données
             vont être effacées ! Continuer
             (O/N) ?');

   10:write('Code article : ');
   11:write('Désignation : ');
   12:write('Prix : ');
   13:write('================================
             ============');
   14:write('Tapez une touche pour
            continuer...');
   20:write(^G,'Article enregistré, numéro :');
   21:write(^G,'Erreur, article non
              enregistré...');
   22:write(^G,'Article inexistant...');
```

```
    23:write(^G,'Modification enregistrée...');
    50:write('Entrez le code à rechercher:');

    99:write(^G,'Erreur, choix inexistant...');
 end;

 If Delai Then
  delay(2000);
 end; {Procédure Message}



 Procedure Affiche;
  {Affiche un article à l'écran}
 begin
    Message(10,3,10,false);
    Write(ArtEnreg.code);

    Message(10,5,11,false);
    write(ArtEnreg.Designation);

    Message(10,7,12,false);
    write(ArtEnreg.prix:1:2);
 end;{ Affiche }


 Procedure Entree;

 {Lit les données d'un article              }
 {Cette procédure est utilisée pour         }
 {la création et la modification d'un       }
 {article                                   }

 var
  temp:str64;
  Res :integer;
 begin
   Message(10,13,10,false);
   Read(temp);
   If temp='' then
    write(ArtEnreg.Code)
   else
    ArtEnreg.Code:=temp;

   Message(10,15,11,false);
```

```
   Read(temp);
   If temp='' then
    write(ArtEnreg.Designation)
   else
    ArtEnreg.Designation:=temp;

   Message(10,17,12,false);
   Read(temp);
   If temp='' then
    write(ArtEnreg.prix:1:2)
   else
    Val(Temp,ArtEnreg.Prix,Res);
end;{ entree}

Procedure Creation;

 {crée le fichier, écrase si il existe déjà}

begin
   If Article.Hdl>0 Then
   begin
    Message(10,20,1,true);
    exit;
   end
   Else
   begin
    Message(5,20,8,False);
    Read(kbd,chh);
    Message(1,20,0,False);
    If Upcase(chh)<>'O' then
    exit;
    Err:=Cree(Article.Nom);
   end;

   Article.hdl:=err;
   If err<0 then
    Message(10,20,3,true)
   else
   begin
    Message(10,20,5,false);
    write(Article.hdl);
    delay(2000);
   end;
end;{Creation}

Procedure Ouverture;
```

```
 {Ouvre le fichier}
begin
  If Article.Hdl>0 Then
  begin
   Message(10,20,1,true);
   exit;
  end
  Else
   Err:=Ouvre(Article.Nom);

  Article.hdl:=err;
  If err<0 then
   Message(10,20,3,true)
  else
  begin
   Message(10,20,6,False);
   write(Article.hdl);
   delay(2000);
  end;
end;{ouverture}

Procedure Fermeture;
 {Ferme le fichier}
begin
  If Article.Hdl=-1 Then
  begin
   Message(10,20,2,true);
   exit;
  end
  Else
   Err:=Ferme(Article.hdl);

  If err=0 then
  begin
   Article.hdl:=-1;
   Message(10,20,7,true);
  end;
end;{Fermeture}


Procedure Cherche;
 {Demande le code d'un article et    }
 {recherche dans le fichier          }
 {                                   }
 {Note:Attention aux majuscules et   }
 {minuscules.....                    }
```

```
Var
 taille:real;
 k,maxrec:integer;
 cle:str12;
begin
  Message(10,1,50,false);
  read(cle);

  taille:=taillefichier(article.hdl);
  MaxRec:=Trunc(Taille/SizeOf(ArtEnreg));
  ok:=False; {assume qu'on ne trouve pas}
  k:=0;
  While not(ok) and (k<Maxrec) Do
  begin
   LitEnreg(Article.hdl,k,SizeOf(ArtEnreg),
   ArtEnreg);
   If ArtEnreg.Code=Cle Then
   begin
    Article.Rnactuel:=k;
    ok:=true; {signale qu'on a trouvé }
    Affiche; {affiche l'article à l'écran}
   end;
   k:=k+1;
  end; {While}
 if not ok then
  message(10,22,22,true);
end; {Procédure Cherche}


Procedure Ajoute;
 {Ajoute un nouvel article dans le fichier}
Var
 RN:Real;
begin
  If article.hdl<0 then
  begin
   Message(10,20,2,true);
   exit;
  end;
  Clrscr;
  Entree; {lit les références de l'article}
  AjouteEnreg(Article.hdl,Rn,SizeOf(ArtEnreg),
              ArtEnreg);

  If ErrCode=0 then
```

```
  begin
    Message(10,24,20,false);
    write(RN:1:0);
    delay(2000);

    MajDir(Article.hdl);
    {Ajuste la taille du fichier dans le }
    {répertoire                          }
   end
   else
    Message(10,24,21,true);
end;{Procédure Ajoute}


Procedure Modifie;
{Recherche d'abord un article et puis  }
{permet de le modifier                 }
begin
 If article.hdl<0 then
 begin
  Message(10,20,2,true);
  exit;
 end;
 Clrscr; {Efface l'écran}
 Cherche;     {Demande le code, recherche }
              {et affiche                 }

 If not Ok then {article pas trouvé...}
  Exit;

 Message(10,10,13,false);

 entree;      {permet les références de }
              {l'article                }
 with article do
  ModifEnreg(hdl,RnActuel,SizeOf(ArtEnreg),
              ArtEnreg);

 Message(10,25,23,True);

end;{Procédure Modifie}


Procedure lit;
 {Lit un article et affiche, utilise la  }
 {procédure CHERCHE                      }
```

```
begin
  If article.hdl<0 then
  begin
    Message(10,20,2,true);
    exit;
  end;

  Clrscr;          {efface l'écran      }
  Cherche;         {demande le code,    }
                   {cherche et affiche  }
  If Not Ok Then {article pas trouvé}
    Exit;

  Message(1,25,14,false);
  Read(kbd,chh);
end;{Procédure Lit}


Procedure Liste;
  {Liste tous les articles du fichier}
var
  taille:real;
  k,maxrec:integer;
begin
  If article.hdl<0 then
  begin
    Message(10,20,2,true);
    exit;
  end;
  Clrscr;{Efface l'écran}
  taille:=taillefichier(article.hdl);
  MaxRec:=Trunc(Taille/SizeOf(ArtEnreg));
  For k:=0 To MaxRec-1 Do
  begin
    LitEnreg(Article.hdl,k,SizeOf(ArtEnreg),
    ArtEnreg);
    With ArtEnreg Do
    Writeln(k:5,':',Code,'*',Designation, '*',
            prix:5:2);
  end;

  Message(1,25,14,false);
  Read(kbd,chh);
end;{Procédure Liste}
```

```
begin {programme}

  {détermine le nom du fichier}
 Article.Nom:='Article.DAT';

  {signale que le fichier n'a     }
  {pas de handle       }
 Article.hdl:=-1;


 Repeat
  Clrscr;
  writeln('MENU');
  writeln('----');
  writeln('(1) Créer un fichier ');
  writeln('(2) Ouvrir un fichier');
  writeln('(3) Fermer un fichier');
  writeln('(4) Ajouter un enreg');
  writeln('(5) Lire un enreg');
  writeln('(6) Modifier un enreg');
  writeln('(7) Liste des enregs');
  writeln('(9) Quitter');
  writeln('');
  write('Votre choix ?');
  Read(kbd,menu);

  Case Menu OF
  '1':Creation;
  '2':Ouverture;
  '3':Fermeture;
  '4':Ajoute;
  '5':lit;
  '6':Modifie;
  '7':Liste;
  end;

  until menu='9';

  {si l'utilisateur a oublié de fermer        }
  {le fichier, on le ferme  }
  If Article.hdl>0 then
   Err:=Ferme(Article.hdl);
end.{Programme}
```

4. MS-DOS et les réseaux locaux

Toutes les fonctions citées sont prévues pour travailler en réseau local, elles sont notamment compatibles avec le système NOVELL.

Si on souhaite limiter l'accès à un fichier ou à une partie de celui-ci, il faudra bloquer la zone en question en utilisant la fonction 5Ch de l'interruption 21h.

Lorsqu'une zone est bloquée ("locked"), toute tentative d'écriture ***et même de lecture*** de cette zone est **interdite** à tous les utilisateurs du réseau, sauf à celui d'où émane l'ordre de bloquage.

Les registres suivants devront être initialisés avant l'appel à cette fonction :

- `Ah` = `5Ch`
- `Al` = `00h` pour bloquer une zone.
  `01h` pour débloquer une zone.
- `Bx` = **Handle.**
- `CX:DX` = Offset du début de la zone à bloquer; cette valeur est exprimée en bytes par rapport au début du fichier.
- `SI:DI` = Longueur de la zone à bloquer; cette valeur est exprimée en bytes.

Au retour, l'indicateur Carry est à 1 s'il y a eu erreur. Si on le souhaite, on pourra appeler la fonction

59h pour obtenir de plus amples informations (voir page 144).

Note : **Avant de refermer le fichier, il faudra débloquer toutes les zones sinon les résultats sont imprévisibles!**

Exemple de programmation

La procédure "RealEnDoubleMot" a été définie ci-dessus (page 142).

```
Procedure
Bloque_debloque( handle    :integer;
                 Pos,
                 Long      :real;
                 Bool      :boolean);

{Bool=true indique que l'on veut bloquer; si }
{bool=false indique que l'on veut débloquer  }

Var
 Ph,Pl:integer;   {Pour coder le réel "Pos"   }
                  {deux integers              }
 Lh,Ll:integer;   {Pour coder le réel "Long"  }
                  {deux integers              }
 Reel:real,

begin
 ErrCode:=0;
 RealEnDoubleMot(Pos,Ph,Pl); {cf. ci-dessus}
 RealEnDoubleMot(Long,Lh,Ll); {cf. ci-dessus}

 With Reg Do
 begin
```

```
    If Bool then
     ax:=$5C01
    else
     ax:=$5C00;
    bx:=handle;
    cx:=ph;
    dx:=pl;
    si:=lh;
    di:=ll;
    intr($21,reg);
    if ((Flags and 1) <> 0) then
      {teste flag CARRY       }
     ErrCode := Ax
    else
     ErrCode := 0
    Erreur(0);
  end;{with reg}
end;{Bloque_debloque}
```

Nous invitons ceux qui envisagent, pour l'adapter aux fonctions réseau, de modifier la "Database Toolbox" de Turbo Pascal à être très prudents. En effet, lorsqu'on souhaite bloquer certaines zones des fichiers d'index, un certain nombre de problèmes se présentent :

- Les index sont gérés par des pages qui sont, autant que possible, gardées en mémoire; il faut donc modifier le code pour que tout changement de clé soit immédiatement écrit sur disque afin qu'il soit accessible aux autres utilisateurs.

- Il est impossible de bloquer une page car ce système est organisé en arbre. Ainsi, comme le bloquage interdit toute opération de lecture, le bloquage d'une page au niveau *i* interdira à tous les autres utilisateurs de lire les pages du niveau *i+1* !!

Il faut donc considérablement modifier le code pour mettre à jour les pages modifiées sur disque. Pour l'autre problème, une solution intéressante consiste à utiliser un fichier annexe dans lequel on ajoute les clés qui doivent être bloquées. Avant toute modification d'une clé, le programme devra vérifier que la clé n'est pas présente dans ce fichier. Si elle s'y trouve, c'est que la clé est déjà en cours de modification et il faut abandonner l'opération. Dès qu'on a fini de modifier une clé, il faut la supprimer de ce fichier. Comme on devra aussi éviter deux accès simultanés à ce fichier, on pourra procéder comme suit :

- Bloquer l'enregistrement 0 du fichier annexe.
- Vérifier que la clé n'existe pas déjà.
- Si oui, abandonner, sinon continuer au point suivant.
- Ajouter la clé.
- Débloquer l'enregistrement 0.

Note : Avant toute modification du code de "Database Toolbox", nous conseillons de l'étudier avec attention car cet algorithme, très efficace, n'est pas parmi les plus simples à comprendre.

5. Limite de 20 fichiers ouverts ?

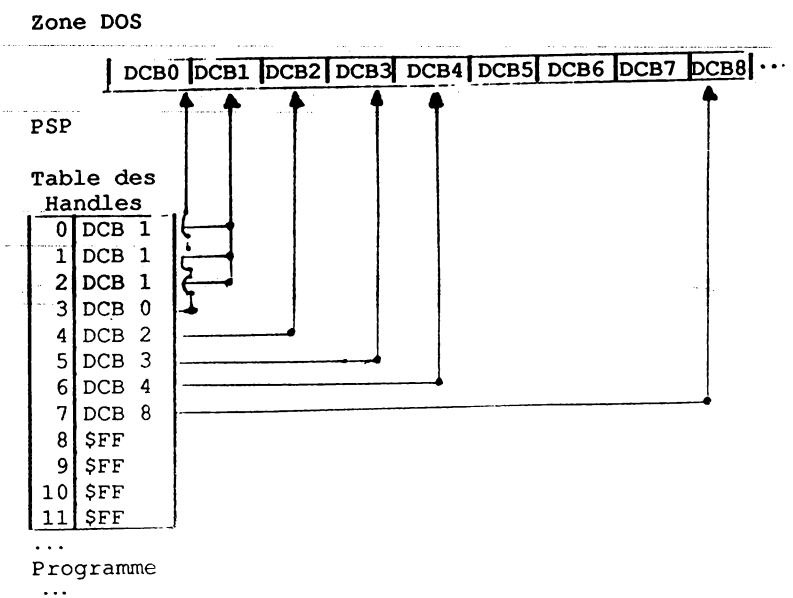
Le système des handles est très souple et bien organisé, mais il possède un inconvénient majeur : dans l'ordinateur, il peut y avoir 96 fichiers ouverts en même temps avec MS-DOS 2.xx et 255 avec MS-DOS 3.xx, mais il ne peut y avoir que 20 fichiers ouverts (dont 5 pour MS-DOS) en même temps **par programme** (l'ordinateur peut avoir plusieurs programmes en mémoire; cf. chapitre 7).

Pour comprendre cela, il faut savoir que MS-DOS gère les fichiers avec des tables appelées **DCB** (device control block), ces tables sont stockées en mémoire dans la zone de données du DOS et le nombre de tables est déterminé par la commande "FILES=..." (CONFIG.SYS). Le nombre maximum de tables est 96 pour MS-DOS 2.xx et 255 pour MS-DOS 3.xx. Le nombre par défaut est 8. Chaque DCB peut être référencé par un numéro (0, 1, 2, 3, 4 ...), les numéros 0, 1 et 2 sont prédéfinis par MS-DOS pour les périphériques AUX, CON et PRN respectivement, les autres sont disponibles pour les fichiers.

Dans le but de supporter la "redirection" des entrées sorties, ces DCB ne sont pas atteintes directement : chaque programme chargé en mémoire (cf. chapitre 7) est précédé d'une zone de 256 bytes appelée PSP qui contient notamment une table de 20 éléments (0.19). **Les éléments de cette table contiendront les numéros des DCB utilisés par le programme**. L'indice de la table (numéro de

l'élément) est le HANDLE dont nous avons parlé au cours des pages précédentes. Les handles 0 à 4 (indices 0.à 4 de la table du PSP) sont réservés pour MS-DOS, les 15 autres éléments de la table seront utilisés pour les fichiers.

Exemple



Si un élément de la table des handles n'est pas utilisé, il contient **$FF**.

Lorsqu'un programme spécifie le handle 6 par exemple, MS-DOS consultera l'élément 6 de la table du PSP et saura qu'il doit utiliser le DCB numéro 4; pour le handle 7 ce sera le DCB 8, etc. Si on spécifie un mauvais numéro de handle (pas attribué), MS-DOS pourra le signaler car il trouve une valeur **$FF** dans

la table. Dès que tous les éléments de la table du PSP sont <> $FF, toute tentative d'ouverture d'un fichier aboutira à un échec; dès que le fichier est fermé, l'élément de la table qui lui était attribué est réinitialisé à $FF.

Comment allons-nous résoudre le problème ? Ce qui nous gêne, c'est le nombre limité d'éléments de la table du PSP; nous allons donc nous occuper de conserver les numéros de DCB dans une table à nous (dans le programme) :

1. Nous créons dans un programme une table de 255 éléments (max).

2. Nous appelons MS-DOS en lui demandant de créer un fichier, il renvoie un handle = 5 (si c'est le premier fichier ouvert dans le programme).

3. Nous allons lire le contenu de l'élément 5 de la table du PSP qui est le numéro du DCB associé au fichier et nous le mettons dans notre table en position 1 (c'est le premier fichier du programme).

4. Nous allons remettre $FF dans l'élément 5 de la table pour libérer le handle.

5. Nous pouvons répéter les points 2, 3 et 4 cent fois si nous le désirons mais en stockant les DCB obtenus dans notre table à la position 1,2,3... jusque 100.

Avec cette méthode, nous avons ouvert 100 fichiers et MS-DOS n'a pas renvoyé d'erreur, ce qui est normal puisqu'il croit qu'il n'y a aucun fichier ouvert : il attribue toujours le premier handle libre qui est toujours 5 étant donné que nous remettons chaque fois $FF dedans après avoir lu et stocké le DCB dans notre table.

Si maintenant nous souhaitons écrire dans le fichier 80 par exemple, il faut faire l'opération inverse :

1- Aller lire le contenu de l'élément 80 de **notre** table (c'est le DCB du fichier 80).

2- Installer ce DCB dans la première position libre de la table du PSP (5).

3- Appeler MS-DOS (fonction écriture) en lui envoyant 5 comme handle.

4- remettre $FF dans la table du PSP (position 5) par sécurité (cas d'un nouvel appel de création de fichier).

On peut aussi répéter cette opération pour nos 100 fichiers en utilisant toujours le **même handle** !!!

Lorsqu'on voudra fermer les fichiers, on utilisera le même procédé.

Avec cette méthode, le nombre maximum de fichiers ouvrables par programme = le nombre maximum de fichiers ouvrables dans l'ordinateur !!

Remarque : **Il faut que le fichier CONFIG.SYS contienne bien la commande FILES=255 pour MS-DOS 3.xx et FILES=96 pour MS-DOS 2.xx car cette commande détermine *le nombre de DCB créés par MS-DOS.***

Exemple de programmation

A titre d'exemple, voici un petit programme qui crée 75 fichiers et les ouvre tous en même temps; il écrit ensuite aléatoirement dans tous les fichiers et puis il les ferme tous. (CONFIG.SYS → FILES=...!!!, ne pas oublier...). Il est intéressant de le recopier (il n'est pas très grand) car il montre bien le principe des handles et des DCB par une animation à l'écran. Ce programme utilise les routines MS-DOS de gestion des fichiers que nous avons vues au cours des pages précédentes, nous les avons regroupées dans le fichier FICH.PAS.

Ce programme utilise les fonctions suivantes de l'interruption **21h** :

- **51h** Lecture de l'adresse du PSP (En version 2.xx, cette fonction, non documentée, est l'équivalent de la fonction **62h** disponible pour les versions 3.xx)

- **Les fonctions de gestion des fichiers (cf. ci-dessus).**

```
Program Handles;

{$F252}
 {cette directive au compilateur doit     }
 {être présente                           }

{$I FICH.PAS}
 {contient les routines DOS de gestion    }
 {des fichiers, cf. chap. précédent       }

Const
 MaxFiles=75;
 MaxLines=200;

Type
 HTable=Array [0..19] Of Byte;

  {Structure pour stocker les informations  }
  {d'un fichier                             }
   FichRec=Record
    Nom : string[64];      {Nom du fichier    }
    hdl : Integer;         {le handle, tjs le }
                           {même...           }
    dcb : byte;            {dcb               }
    RnActuel: integer;
   end;


Var
  message     :string[80];
  buffer      :array[1..80] of char;
  NosFichiers :array[1..Maxfiles] of FichRec;
  Err         :integer;
  rn          :real;
  chh         :char;
  HandleTable :^HTable;
  handle      :integer;
  I,N,
  MaxOuverts  :Integer;
  Done        :Boolean;
  NomFich     :string[64];
```

```
  reg          :registres;

Procedure AjusteHandle(Numfich: Integer);
  {remet le numéro de DCB dans la table      }
  {des handles le handle est toujours le     }
  {même....                                  }

Begin
  Handle:=NosFichiers[Numfich].hdl;
  HandleTable^[handle]:=
    NosFichiers[Numfich].dcb;
End;{AjusteHandle}


Procedure GetHandleTable;
  {cette procédure ajuste notre variable     }
  {HANDLETABLE pour qu'elle soit au même     }
  {emplacement mémoire que la table du PSP,  }
  {ce qui nous permettra de lire facilement  }
  {les informations du PSP...                }

Begin
  With Reg Do
  Begin
    AH:=$51;    {Cette fonction permet          }
                {d'obtenir l'adresse            }
                {du PSP,                        }
                {il en existe une               }
                {autre mais pour DOS 3.xx       }
                {qui fait exactement la         }
                {même chose, celle-ci est       }
                {non documentée.                }
    MsDos(Reg);
                {ajuste le pointeur HandleTable }
                {vers la table du PSP           }
                {offset 18h, cf. chap. 4, le PSP}
    HandleTable:=Ptr(BX,$18);
  End;
End;{GetHandleTable}

Procedure ShowHandleTable;

  {Petite procédure d'animation écran qui      }
  {montre bien qu'on n'utilise qu'un seul handle}
  {pour les 75 fichiers...                     }
```

```
Var
 I: Integer;
Begin
 For I:=0 To 19 Do
 Begin
  GotoXY(1,I+2);
  Write(HandleTable^[I]);
  if HandleTable^[I]=255 then
   write(' Inutilisé...');
  if i=5 then
  begin
   clreol;
   write(' <-- seul ce handle est utilisé
        pour tous les fichiers!!!');
  end;
 End;
 delay(120);     {sinon on ne voit rien, ça  }
                 {va trop vite...            }
End;{ShowHandleTable}

Begin
 ClrScr;
 GetHandleTable;
 writeln('Table des Handles dans le PSP:');
 writeln;
 ShowHandleTable;      {Affiche la table des }
                       {handles du PSP       }
 I:=1;
 Done:=False;
 While (I<=MaxFiles) And Not Done Do
 Begin
   GotoXY(1,24); ClrEOL;
   Write('Ouverture du fichier #',I);

   {crée FICH.000, FICH.001, etc.: }
   NomFich:='FICH.'+
   Chr(48+I Div 100)+
   Chr(48+(I Mod 100) Div 10)+
   Chr(48+(I Mod 10));

   NosFichiers[i].Nom:=Nomfich;
   Handle:=cree(nomfich);
   ShowHandleTable;      {Affiche la table    }
                         {des handles du PSP  }
   if handle<>-1 then
   begin
```

```
    NosFichiers[i].hdl:=handle;
    NosFichiers[i].dcb:=Handletable^[Handle];
    HandleTable^[Handle]:=$FF;
     {vide le handle de la table          }
     {pour qu'il puisse être à            }
     {nouveau attribué                    }
    GotoXY(1,23); ClrEOL;
    with nosfichiers[i] do
     writeln('Fichier [',i,'] ,
              handle:',handle,' dcb:',dcb,' ',
              nomfich);
    writeln;
    I:=I+1;
   end
  else
  {erreur, nombre de fichiers max atteint }
  {dépendant de FILES=...dans CONFIG.SYS }
   done:=true;
End;{while}

MaxOuverts:=I-1;
GotoXY(1,23); ClrEOL;
GotoXY(1,24); ClrEOL;
WriteLn(MaxOuverts,' Fichiers ouverts en même
        temps...');

For N:=1 To MaxLines Do
 {écrit des données aléatoirement  }
 {dans les fichiers                }
Begin
 I:=Random(MaxOuverts)+1; {Choisi un   }
                          {fichier au  }
                          {hasard      }
 GotoXY(1,25); ClrEOL;
 Write('Ecriture de la ligne #',N,' dans le
       fichier #',I);
 AjusteHandle(I);
 ShowHandleTable;       {montre que l'on     }
                        {utilise 1 handle    }

 {crée le message à mettre dans le fichier,   }
 {ajoute CR et LF pour que la prochaine       }
 {écriture soit à la ligne suivante et        }
 {transfère dans un buffer pour ne pas écrire }
 {message[0] qui contient la longueur de la   }
 {chaîne de caractère                         }
```

```
  str(i,message);
  message:='Ceci est le fichier #'+
  message+#13+#10;
  move(message[1],buffer,length(message));

  With NosFichiers[i] Do
   AjouteEnreg(hdl,Rn,length(Message),
               buffer);
 End;{for}

 GotoXY(1,25); ClrEOL;
 For I:=1 To MaxOuverts Do
  { On ferme tous les fichiers }
 Begin
  GotoXY(1,24); ClrEOL;
  Write('Fermeture du fichier #',
        I,'',nosfichiers[i].nom);
  AjusteHandle(I);
  ShowHandleTable;   {Montre le contenu de    }
                     {la table des handles    }
  Err:=Ferme(NosFichiers[i].hdl);
 End;

 clrscr;
 writeln;
 writeln;
 writeln;
 writeln('Utilisez la commande TYPE pour
          visualiser le contenu des
          fichiers...');

End.{Programme}
```

Chapitre 6

Organisation des fichiers sur disque

1. Introduction

Dans ce chapitre, nous allons voir ce qui se cache derrière les répertoires : qu'est-ce qu'un secteur, une piste, un cylindre,...? Comment les fichiers sont-ils stockés sur disque? Comment est-il possible de récupérer un fichier effacé ? Comment protéger une disquette contre la copie? Que font les programmes comme CHKDSK, NORTON UTILITIES, PCTOOLS, FDISK ou autres programmes de diagnostic ?

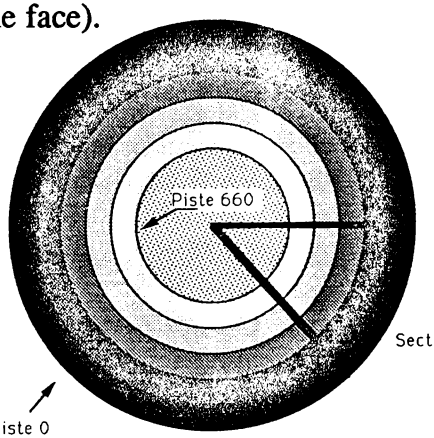
Si l'envie vous vient de réécrire un de ces programme, nous verrons que cela n'est pas si compliqué que ça en a l'air. Mais attention, vous pouvez programmer tout ce que vous souhaitez sans danger si vous ne faites que des opérations de lecture. Au pire, vous risquez de "planter" la machine, ce qui n'est pas bien grave. Si, par contre, vous écrivez des informations sur disque, faites d'abord un BACKUP car on se trompe vite d'une instruction et cela peut causer de très gros dégâts. Avec un backup, dans le pire des cas, il faudra reformater et restaurer la copie. Notons aussi que lors de la mise au point de programmes de ce type, il est conseillé de les tester sur des disquettes et de ne passer sur disque dur que quand le programme est parfaitement au point. Donc **prudence**, personne n'est à l'abri d'une petite erreur d'inattention !!!!

2. Structure du disque.

Un disque (disquette ou disque dur) est un périphérique orienté "bloc de caractères" divisé en un certain nombre de **secteurs** (cf. le chapitre sur les périphériques : les périphériques bloc).

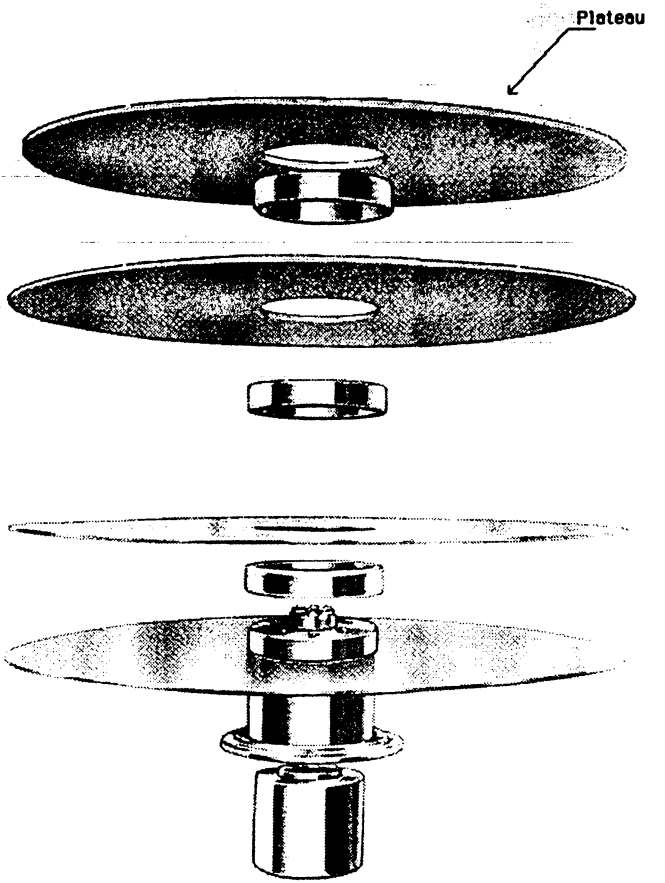
> **Un secteur est l'unité de base d'accès à un disque; il contient 512 bytes.**

Le disque est constitué d'un ou de plusieurs "plateaux" (1 pour les disquettes), chaque plateau peut avoir 2 faces, chaque face est constituée d'une série de cercles concentriques appelés **pistes** (tracks); chaque piste peut contenir 8 ou 9 secteurs, cela dépend de la version de DOS utilisée (les disquettes peuvent n'utiliser qu'une seule face).



Le nombre de pistes est une des caractéristiques du disque dur.
Chaque piste est divisée en un nombre de secteurs dépendant du formatage (8 à 15).

L'ensemble des pistes numéro "x" de chaque plateau est appelé cylindre :



Le nombre de plateaux est une des caractéristiques du disque dur.

3. Analyse du disque.

Comme le disque est un ensemble de secteurs, on pourra accéder à chacun d'eux individuellement. Les deux méthodes d'accès à un secteur sont les suivantes :

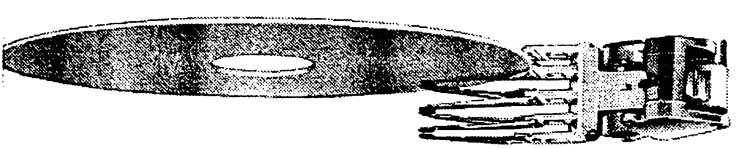
1. La numérotation **réelle** qui requiert les informations suivantes :

   - Un numéro de **piste** (ou cylindre).
   - Un numéro de **face** (tête de lecture).
   - Un numéro de **secteur** dans la piste.

2. La numérotation **logique** qui requiert :

   - Un numéro de **secteur**, appelé secteur "logique". Ce numéro, exprimé à partir du premier secteur du disque, sera converti sous la forme "piste, face, secteur" par MS-DOS.

La conversion est bien entendu liée à des considérations matérielles : les "têtes" de lecture/écriture. Elles sont toutes attachées à une sorte de peigne et lorsqu'on spécifie un numéro de piste, ce peigne sera ajusté pour se trouver à hauteur du cylindre correspondant.



Les têtes de lecture sont attachées à une sorte de peigne.

On peut donc en déduire comment se fait la conversion de la méthode 2 vers la méthode 1 : il faut minimiser le déplacement des têtes (peigne) car cela prend du temps. La numérotation logique est donc la suivante : le peigne commence sur la piste 0 (cylindre 0), le premier secteur logique (= secteur 0) sera le premier secteur (=1) de cette piste, il sera atteint par la tête 0, la numérotation se poursuivra jusqu'au dernier secteur de la piste (8 ou 9) et ensuite on passera à la tête 1 puis 2, ... le peigne n'est pas déplacé, quand on sera arrivé à la dernière tête, on avancera le peigne sur le cylindre suivant et on recommencera avec la tête 0, etc.

Notes: Par abus de langage, on appelle parfois le numéro de tête "face" car, pour chaque plateau, il y a une tête de lecture/écriture par face.

De même, le numéro de piste est aussi appelé "cylindre" car l'ensemble des pistes numéro "x" de tous les plateaux est un cylindre et que le numéro de tête permet d'indiquer le plateau et la face souhaitée; il n'y a donc aucune confusion possible.

4. Les interruptions Disque

Il existe des interruptions permettant d'utiliser l'une ou l'autre méthode, logique ou réelle:

- Interruption **13h** Accès disque avec numérotation réelle (lecture, écriture, formatage,...).

- Interruption **25h** Lecture de secteurs avec numérotation logique.

- Interruption **26h** Ecriture de secteurs avec numérotation logique.

L'interruption 13h - Gestion du disque

L'interruption 13h est très complète, elle propose les fonctions suivantes :

00h **Réinitialisation du disque (à appeler lorsqu'une erreur s'est produite).**

01h **Lecture du *status* du disque (indique le résultat de la dernière opération).**

02h **Lecture de secteurs (numérotation réelle, il faudra spécifier la piste, la tête et le secteur).**

03h Ecriture de secteurs (numérotation réelle,...).

04h Vérification de secteurs (numérotation réelle,...).

05h Formatage de pistes.

08h Lecture des paramètres du disque (Floppy ou disque dur, nombre de secteurs par piste, nombre de cylindres,...).

0Ch Positionnement du peigne sur un cylindre mais il n'y a ni lecture ni écriture.

10h Teste si le disque est prêt.

Les interruptions 25h et 26h — Lecture et écriture logique de secteurs

Les interruptions **25h** et **26h** ne disposent pas de fonctions; on peut les appeler pour lire ou écrire des secteurs logiques. Pour les utiliser, il faudra spécifier :

- Le numéro du disque (0='A:', 1='B:', 2='C:',...) dans le registre **AL**.

- Le nombre de secteurs à lire ou écrire dans le registre **CX**.

- Le secteur départ de l'opération dans le registre **DX**.

- L'adresse d'une zone où les données seront lues/transférées dans la paire de registres **DS:BX**.

Au retour, l'indicateur carry est à 0 si tout s'est bien passé, et il est à 1 si une erreur s'est produite. Dans ce dernier cas, **AX** contient un code d'erreur :

- **AL** contient un code identique à ceux renvoyés dans DI par l'interruption **24h** (cf. chapitre sur les périphériques).

- **AH** contient une des valeurs suivantes :

  - **80h** Le périphérique ne répond pas.

- **40h** Erreur de positionnement des têtes de lecture/écriture.
- **20h** Erreur du contrôleur.
- **10h** Erreur de donnée.
- **08h** Erreur de transfert (DMA).
- **04h** Secteur non trouvé.
- **02h** Mauvaise adresse.
- **01h** Mauvaise commande.

Exemple de programmation

Nous allons élaborer, petit à petit, tout un programme de gestion du disque. Le premier élément est une petite procédure en Turbo Pascal incorporant du langage assembleur qui appelle l'interruption **25h** (lecture):

```
Procedure
Int25(drive      :Byte;  {Disque}
      LSN        :Integer;{Premier secteur à lire}
      sects      :Integer;{Nbr de sect. à lire}
      VAR buffer;{buffer recevra le
                  secteur lu}
      VAR err    :Integer);{code d'erreur}
{-------------------------------------------}

{Procédure de lecture d'un secteur logique sur
disque, fait appel à l'interruption 25h  }

Begin
 INLINE(
 $55/           { PUSH  BP          }
 $1E/           { PUSH  DS          }
 $8A/$46/$10/   { MOV   AL,[BP+10]  }  {Al<-drive}
 $8B/$56/$0E/   { MOV   DX,[BP+0E]  }  {DX<-LSN}
 $8B/$4E/$0C/   { MOV   CX,[BP+0C]  }  {CX<-Sects}
 $C5/$5E/$08/   { LDS   BX,[BP+08]  }  {DS:BX<-
                                        buffer}
```

```
    $CD/$25/        { INT   25          }
    $72/$02/        { JB    Suite       } {erreur?}
    $31/$C0/        { XOR   AX,AX       } {non,Ax<--0}
    $9D/            { Suite: POPF       } {flags orig}
    $1F/            { POP   DS          }
    $5D/            { POP   BP          }
    $C4/$7E/$04/    { LES   DI,[BP+04]  }
    $26/            { ES:               }
    $89/$05         { MOV   [DI],AX     } {Err <--AX}
    );
  End; {int 25}
```

Note : On ne peut malheureusement pas utiliser la procédure "Intr" de Turbo Pascal car l'interruption **25h** renvoie des résultats via le registre *flags*. Celui-ci n'est donc pas retiré de la pile et c'est le programme qui, pour conserver la stabilité de la pile, doit s'en charger.

5. Découpage d'un disque dur

Un disque dur peut être découpé en plusieurs unités logiques, on parlera de **partitionner** le disque. Cela permettra notamment d'utiliser différents systèmes d'exploitation au sein d'un même ordinateur; il faudra cependant spécifier quel est le système d'exploitation qui sera chargé au démarrage de l'ordinateur : c'est la partition **"boot"**. Au sein de chaque unité, les secteurs sont organisés comme s'il s'agissait d'un disque dur indépendant. Nous pouvons maintenant préciser certaines notions que nous avons déjà énoncées.

- La numérotation "**réelle**" ou "physique" des secteurs est exprimée sous la forme **tête-secteur-piste**; chaque secteur du disque possède une et une seule référence de cette forme.

- La numérotation "**logique**" des secteurs commence à 0, à partir du premier secteur physique de chaque unité.

Il pourra donc y avoir 2 secteurs logiques 0 dans un disque dur (1 pour chaque unité) mais il n'y aura jamais qu'un seul secteur physique 0.

Pour implanter ce système, on a déterminé que le premier secteur **physique** du disque contiendra les informations sur les partitions, il est organisé comme suit :

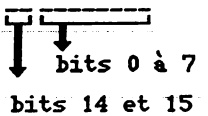
Offset depuis le début du secteur	taille (en bytes)	description
01BEh	16	Partition 4
01CEh	16	Partition 3
01DEh	16	Partition 2
01EEh	16	Partition 1
01FEh	2	signature (doit être AA55h)

Pour chaque partition, il y a un ensemble de 16 bytes qui contient les informations suivantes :

Offset	Taille	Description
00h	1	La partition est "bootable" (80h) ou non (00h).
01h	1	Tête du premier secteur du disque associé.
02h	2	Numéro du premier secteur et cylindre du disque (*).
04h	1	01h indique que la fat est à 12 bits, 04h indique qu'elle est à 16 bits.
05h	1	Tête du dernier secteur.
06h	2	Numéro du dernier secteur et cylindre (*).
08h	4	Numéro du premier secteur par rapport au début du disque.
0Ch	4	Nombre de secteurs de cette partition.

Notes :
- **Il ne peut y avoir qu'une partition "bootable".**
- **Le travail d'un programme comme FDISK consiste donc à modifier le premier secteur du disque.**

(*) - Le numéro de cylindre est une valeur sur 10 bits composée comme suit:



- Le numéro des secteurs est codé sur 6 bits (bits 8..13 du mot).

Exemple de programmation

Voici comment lire et interpréter le secteur contenant les informations sur les partitions :

```
Program Partitions;
type
 str4=string[4];

 {définition du contenu de la zone de 16 bytes}
 parti=record
  bootindic:byte;
  beghead:byte;
  begsect:byte;
  begcyl:byte;
  sysindic:byte;
  endhead:byte;
  endsect:byte;
  endcyl:byte;
  startsectl:integer;
  startsecth:integer;
  nbsectl:integer;
  nbsecth:integer;
 end;{parti}
```

```
    sectrec=array[0..512] of char;

  Var
    i,sign:integer;
    disque:byte;
    partitions:array[1..4] of parti;
    buffer:sectrec;

  Function ErrDisque:boolean;
    {regardre s'il y a eu erreur lors de   }
    {de la dernière opération disque       }
  begin
    with reg do
    begin
      ah:=1;
      intr($13,reg);
      if al=0 then
        ErrDisque:=false
      else
        errdisque:=true;
    end;
  end;{errDisque}

  Procedure ResetDisque(disque:byte);
  begin
    with reg do
    begin
      ah:=0;
      dl:=disque;
      intr($13,reg);
    end;
  end;


  Function Int2Hex4(k:integer):str4;
    {transforme une valeur en hexa}
  const
    Hexa:array[0..15] of char='0123456789ABCDEF';
begin
  Int2Hex4:=  Hexa[(k shr 12)]+
              Hexa[(k shr 8) and 15]+
              Hexa[(k shr 4) and 15]+
              Hexa[(k and 15)];
end;{int2Hex4}
```

```
begin {programme}

 disque:=$80;{disque dur}
 ResetDisque($80);
 with reg do
 begin
  ax:=$0201;{lire 1 secteur}
  cx:=1;{cyl 0, sect 1}
  dh:=0;{tete}
  dl:=disque;
  es:=dseg;
  bx:=offset(buffer);
  intr($13,reg);
 end;{with reg do}

 if ErreurDisque then
 begin
  writeln('erreur de lecture...');
  halt;
 end;

  {copie les infos des 4 partitions dans    }
  {notre array:                             }
 Move(buffer[$01BE],partitions,64);

  {copie la signature dans "sign":}
 Move(Buffer[$01FE,Sign,2);

 If Int2Hex4(Sign)<>'AA55' then
 begin
  writeln('Pas de signature, ce secteur ne
          contient pas d'informations sur les
          partitions');
  halt;
 end;

 For i:=1 to 4 do
  with partitions[i] do
  begin
   writeln;
   write('Partition ',i);
   write;
   if bootIndic=$80 Then
    writeln(' bootable')
   else
    writeln(' Non bootable');
```

```
    writeln;
     {cf remarque ci-dessus}
    Sect:=Begsect and $3F;          { 3F = 00111111}
    Cyl:=(BegSect and $C0) shl 2;{ C0 = 11000000}
    Cyl:=Cyl+Begcyl;
    writeln ('Début:',beghead:4,' ',sect:4,' ',
             cyl:4);
    Sect:=Endsect and $3F;
    Cyl:=(EndSect and $C0) shl 2;
    Cyl:=Cyl+Endcyl;
    writeln ('Fin:',endhead:4,' ',sect:4,' ',
             cyl:4);
  end;{with}
end;{partitions}
```

6. Organisation du Disque

La partie DOS (partition DOS pour un disque dur, le disque tout entier pour une disquette) est divisée de la manière suivante :

a. Zone de démarrage (boot).
b. Table d'allocation (FAT 1).
c. Table d'allocation (FAT 2, c'est une copie de la FAT 1).
d. Répertoire principal (Root, « \ »,...).
e. Zone pour les fichiers.

La taille de ces éléments dépend d'un certain nombre de paramètres (disquette ou disque dur, simple face ou double face, version du DOS, ...), elle est initialisée au formatage et ne sera jamais modifiée par la suite.

- L'élément de base d'accès à un disque est le **secteur.**

- l'unité d'allocation d'espace disque à un fichier est appelée "**cluster**".

- Un **cluster** est constitué d'un ou de plusieurs secteurs, cette valeur est déterminée au formatage.

- Tous les clusters d'un fichier sont répertoriés dans la **FAT** (File Allocation Table).

La zone de démarrage (boot record)

Cette zone est située sur la piste 0, secteur 1, face 0, tête 0 de toutes les disquette DOS; pour les disques durs, c'est le premier secteur de la partition DOS (nous y reviendrons). Il contient les informations suivantes :

Bytes	Signification
0.. 2	Instruction de saut au programme de démarrage (boot).
3..10	Nom du constructeur.
11..12	Nombre de bytes par secteur.
13	Nombre de secteurs par cluster.
14..15	Nombre de secteurs réservés au début de la partition (si disque dur) ou au début de la disquette.
16	Nombre de FATs.
17..18	Nombre max. de fichiers dans le répertoire principal.
19..20	Nombre total de secteurs sur disque.
21	Description du disque :
	FFh disquette double face, 8 secteurs/piste.
	FEh disquette simple face, 8 secteurs/piste.
	FDh disquette double face, 9 secteurs/piste.
	FCh disquette simple face, 9 secteurs/piste.
	F9h disquette double face, 15 secteurs/piste.
	F8h disque dur.
22..23	Nombre de secteurs utilisés par une FAT.
24..25	Nombre de secteurs par piste.
26..27	Nombre de têtes de lecture/écriture.
28..29	Nombre de secteurs cachés.

Exemple de programmation

Ces informations seront nécessaires pour accéder aux répertoires et aux fichiers. En utilisant la procédure de l'interruption 25h, nous pouvons commencer notre programme en lisant et en affichant les paramètres contenus dans cette zone:

```
Program BootFatDir;
Const

 Root =true;
 noroot=false;
 TailleSect=512;{nombre de bytes d'un secteur}


Type

 str12=string[12];

 {pour stocker les param. lus dans le Boot }
 {record "BR"=Boot Record, "ofs"=Offset    }

ParamDisque=Record
   BytesPSecteur:Integer; {ofs 11 du BR, 2 bytes }
   SectsPCluster:Integer; {ofs 13 du BR, 1 byte  }
   SectsReserves:Integer; {ofs 14 du BR, 2 bytes }
   NbrDeFats :Integer;    {ofs 16 du BR, 1 byte  }
   MaxEntrees :Integer;   {ofs 17 du BR, 2 bytes }
   TotSects :Real;        {ofs 19 du BR, 2 bytes }
   media :Integer;        {ofs 21 du BR, 1 byte  }
   SectsPFat :Integer;    {ofs 22 du BR, 2 bytes }
   SectsPPiste :Integer;  {ofs 24 du BR, 2 bytes }
   Nbrtetes :integer;     {ofs 26 du BR, 2 bytes }
   PremFat :Integer;      {premier sect. de la   }
                          {fat, calculé          }
   PremRoot :Integer;     {premier sect. de la   }
                          {dir, calculé          }
   SectsPRoot :Integer;   {nbr de sect. de la    }
                          {dir, calculé          }
   Premdata :Integer;     {premier sect. des     }
                          {fichiers, calculé     }
   TotPistes :Integer;    {Nbr de pistes,        }
```

```
                                {calculé                }
    TotClusters :Integer;       {Nbr de clusters (pour  }
                                {les fichiers)          }
    BitsdeFat :byte;            {Nbr de bits d'un       }
                                {élément de la FAT      }
  end;{param Disque}


 var

  Disque  :byte;          {Num du disque          }
  Params  :ParamDisque;   {Paramètres du disque   }

{en fct du niv. du répertoire, doit être global}
  Indent  :string[80];

 PROCEDURE
 Int25( drive       :Byte;      {Disque             }
        LSN         :Integer;   {Secteur à lire     }
        sects       :Integer;   {Nbr de sect à lire }
        VAR buffer;             {buffer recevra     }
                                {le secteur lu      }
        VAR erreur :Integer); {code d'erreur        }
{-------------------------------------}

 {Procédure de lecture d'un secteur logique    }
 {sur disque, fait appel à l'interruption 25h  }
 Begin

 {... cf. pages précédentes}

 End; {int 25}


 Function Char2EnInteger(a,b:char):Integer;
 {-------------------------------------}
 {traduit 2 char (code ascii) en 1 nbr entier}

 begin

 Char2EnInteger:=ord(b)*256+ord(a); {inversion   }
                                    {des bytes...}

end;  {Char2EnInteger}
```

```
Function Char2EnReal(a,b:char):real;
{-----------------------------------}
{traduit 2 char (code ASCII) en 1 nbr réel}

begin

 Char2EnReal:=ord(b)*256.0+Ord(a);
 {inversion des bytes...}

end;  {Char2EnReal}

 Procedure LitBootRecord(drive:byte);
 {-----------------------------------}
 var
  erreur :Integer;
  Buffer :array[0..TailleSect] of char;
  {buffer pour int25}
  tempreal :real;
 begin

  {lecture de secteur à partir du secteur 0}
  Int25(drive,0,1,Buffer,Erreur);
  If Erreur<>0 Then
  begin
   writeln('erreur de lecture sur le secteur
            BOOT');
   halt;
  end;

  With Params Do
  begin
   BytesPSecteur := Char2EnInteger(Buffer[11],
                    buffer[12]);
   SectsPCluster := Ord(buffer[13]);
   SectsReserves := Char2EnInteger(Buffer[14],
                    buffer[15]);
   NbrDeFats:= Ord(buffer[16]);
   MaxEntrees:=Char2EnInteger( Buffer[17],
                               buffer[18]);
   TotSects :=Char2EnReal( Buffer[19],
                           buffer[20]);
   media := ord(buffer[21]);
   SectsPFat := Char2EnInteger( Buffer[22],
                                buffer[23]);
```

```
    SectsPPiste:=Char2EnInteger( Buffer[24],
                                 buffer[25]);
    NbrTetes:=Char2EnInteger( Buffer[26],
                              buffer[27]);
    PremFat:= SectsReserves;
    PremRoot:= SectsReserves+( NbrDeFats*
                               SectsPFat);
    SectsPRoot:=(Maxentrees*32)Div BytesPSecteur;
    If ((Maxentrees*32) mod BytesPSecteur)<>0 Then
     SectsPRoot:=SectsPRoot+1;

    premdata := PremRoot+SectsPRoot;

    tempreal:= Totsects/
    (Nbrtetes*SectsPPiste);
    totpistes := trunc(Tempreal);

    If TempReal-Totpistes>0.0 Then
     Totpistes:=totpistes+1;

    tempreal:=(totsects-premdata)/SectsPCluster;
    TotClusters:= trunc(TempReal);

    {cf. chap. suivant, la FAT...}
    If TotSects>4086 Then
     BitsdeFat:=16
    Else
     BitsdeFat:=12;
  End;{with params}
end; {LitBootRecord}


Procedure AfficheDiskParams;
{-------------------------}
begin

 {affiche, '\'=répertoire principal...}
 With Params Do
 begin
  Writeln('Nombre de bytes par secteur:',
          BytesPSecteur );
  Writeln('Nombre de secteurs par Cluster: ',
          SectsPCluster );
  Writeln('Nombre de secteurs réservés:',
          SectsReserves ,' (pour le BR)');
  Writeln('Nombre de FATs: ',NbrDeFats );
```

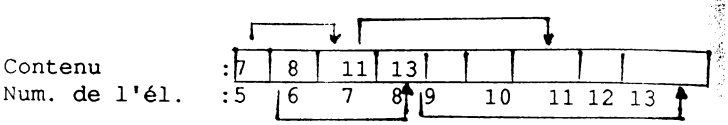
```
  Writeln('Nombre d'entrées dans le "\":',
          MaxEntrees );
  Writeln('Nombre total de secteurs:',
          TotSects:1:0);
  Writeln('Nombre de secteurs par FAT:',
          SectsPFat );
  Writeln('Nombre de secteurs par piste:',
          SectsPPiste );
  Writeln('Nombre de têtes (faces):',
          NbrTetes );
  Writeln('Premier secteur de la FAT:',
          PremFat );
  Writeln('Premier secteur de "\":',PremRoot);
  Writeln('Nombre de secteurs de "\":',
          SectsPRoot);
  Writeln('Premier Secteur des fichiers:',
          premdata );
  Writeln('Nombre de pistes (cylindres):',
          totpistes );
  Writeln('Nombre de clusters (pour fichiers):',
          TotClusters);
  Writeln('Longueur des éléments de la FAT:',
          bitsdeFat,' bits');
  writeln;
  case ord(media) of
   $FF:writeln(' --> Disquette double face,
                     8 secteurs/piste ');
   $FE:writeln(' --> Disquette simple face,
                     8 secteurs/piste ');
   $FD:writeln(' --> Disquette double face,
                     9 secteurs/piste ');
   $FC:writeln(' --> Disquette simple face,
                     9 secteurs/piste ');
   $F9:writeln(' --> Disquette double face,
                     15 secteurs/piste');
   $F8:writeln(' --> Disque dur');
  end;
 end;{With Params}
end;{affiche DiskParams}

begin {program}
 clrscr;
 disque:=2; {0='A:',1='B:',2='C:',...}

 AfficheDiskParams;
end. {program}
```

La FAT (table d'allocation des fichiers).

Nous avons vu que l'unité de base d'allocation d'espace disque était le "cluster", la FAT est utilisée par MS-DOS pour attribuer des clusters libres à un fichier et pour connaître la liste des clusters appartenant à un fichier. Il s'agit en fait d'une grande table contenant autant d'éléments qu'il y a de clusters sur le disque.

Structure de la FAT

Le principe est que chaque élément de cette table contient le numéro d'un autre élément, ce qui crée une liste de clusters. Par exemple :



Dans ce cas :

- L'élément 5 pointe vers l'élément 7 (l'élément 5 contient la valeur 7).

- L'élément 6 pointe vers l'élément 8 (l'élément 6 contient la valeur 8).

- L'élément 7 pointe vers l'élément 11 (l'élément 7 contient la valeur 11).

- l'élément 8 pointe vers l'élément 13 (l'élément 8 contient la valeur 13).

- etc.

Pour chaque fichier, il existe une liste de ce type liant entre eux les clusters contenant le fichier. Les répertoires (cf. ci-dessous) contiennent, pour chaque fichier, une zone de 32 bytes : c'est dans cette zone que l'on trouvera notamment le numéro du premier cluster du fichier indiquant le début de la liste dans la FAT. La fin de la liste est repérée lorsque l'élément contient une des valeurs entre FFF8h et FFFFh.

Il est aussi possible que certains clusters soient mauvais (erreur de lecture ou écriture quand on y accède), il faudra alors les "marquer" pour qu'ils ne soient pas attribués à un fichier; il faudra aussi marquer les clusters qui sont libres. Un élément de la FAT pourra donc contenir les valeurs suivantes :

Valeur	Signification
0000h	Indique que le cluster est libre.
FFF7h	Indique que le cluster est mauvais.
FFF8h..FFFFh	Indique que c'est le dernier cluster d'un fichier.
Autre	Pointeur vers élément suivant dans la liste.
0001h	Valeur impossible (réservée).

Les Fats à 12 et 16 bits

Ces valeurs, codées sur 4 chiffres hexadécimaux, indiquent qu'un élément de la FAT est codé sur 2 bytes (16 bits). Pour simplifier les choses (?), il existe un deuxième type de FAT où les éléments sont sur 12 bits; c'est pour économiser de la place sur les disques où il y a moins de clusters (16 bits ==> jusque 2 puissance 16 clusters, 12 bits ==> jusque 2 puissance 12 clusters...). En fait, si le disque a moins de 4086 clusters, MS-DOS utilise une FAT à 12 bits et à 16 bits autrement. Les codes disponibles dans des FATs à 12 bits (on supprime le premier signe hexa) sont les suivants :

Valeur	Signification
000h	Indique que le cluster est libre.
FF7h	Indique que le cluster est mauvais.
FF8h..FFFh	Indique que c'est le dernier cluster d'un fichier.
Autre	Pointeur vers élément suivant dans la liste.
001h	Réservé.

Notons encore que les deux premiers éléments de la FAT sont réservés, le premier byte de la FAT est **l'identificateur de média** :

```
FFh -->    disquette double face, 8
           secteurs/piste
FEh -->    disquette simple face, 8
           secteurs/piste
FDh -->    disquette double face, 9
           secteurs/piste
```

```
FCh -->     disquette simple face,  9
            secteurs/piste
F9h -->     disquette double face,  15
            secteurs/piste
F8h -->     disque dur
```

Les 2 ou les 3 bytes qui suivent (2 si FAT à 12 bits, 3 si FAT à 16 bits) sont = **FFh** (12bits), **FFFh** (16bits). Le premier cluster pouvant être attribué à un fichier est donc le cluster 2 (nous verrons plus loin comment convertir un numéro de cluster en un numéro de secteur et inversement).

Exemple de programmation

Nous pouvons maintenant écrire une procédure qui lit la FAT : son premier secteur et le nombre de secteurs sont connus depuis la lecture de la zone de démarrage (cf. pages précédentes).

Nous devons d'abord ajouter la variable suivante à notre programme :

```
var

 FAT:ARRAY[0..32767] OF Byte;
 {Copie de la FAT en mémoire}
```

Et ensuite la procédure :

```
 Procedure
 LitFat(Drive,PremierSect,NbrSect:Integer);
{--------------------------------------}
```

```
var
 erreur,i,j :Integer;
 Secteur :Integer;
 Buffer :Array[0..TailleSect] of char;
 {buffer pour int25}

begin
 Secteur:=PremierSect;
 i:=0;
 j:=0;
 while (i<NbrSect) Do
 begin
  Int25(drive,Secteur,1,Buffer,Erreur);
  If Erreur<>0 Then
  begin
   writeln('erreur de lecture de la FAT');
   halt;
  end;

  Move(Buffer,Fat[j],512);
  j:=j+512;

  Secteur:=Secteur+1;
  i:=i+1;
 end; {while i}
end;{LitFat}
```

Pour l'appel, les lignes suivantes seront insérées après la lecture de la zone de démarrage :

```
With Params Do
 begin
  LitFat(Disque,PremFat,SectsPFat);
 end;
```

Consultation d'une FAT

Comme 12 bits équivaut à un byte et demi et que la plus petite unité adressable par le microprocesseur 8088 est le byte (8 bits), la méthode de consultation d'une FAT à 12 bits est beaucoup plus compliquée que pour une FAT à 16 bits. Dans une fat à 16 bits, pour lire l'élément 4, par exemple, il suffira de calculer 4*2; on obtiendra alors le déplacement dans la fat et on pourra lire les 2 bytes qui sont là. Dans une FAT à 12 bits, il y a possibilité de mettre 3 signes hexadécimaux par élément (1 signe Hexa = 4 bits, cf. conversion binaire/hexadécimale), certains éléments seront donc en début d'un byte (les éléments pairs, en commençant à partir de 0), d'autres pas (les éléments impairs); la méthode d'extraction apparaît donc relativement compliquée. Dans le but de faciliter la programmation en assembleur, et donc d'augmenter la rapidité de consultation d'un élément, une technique un peu compliquée a été implantée.

Imaginons que le contenu d'un élément soit ABC (valeur hexa) et le contenu de l'élément suivant DEF (3 signes hexa), on pourrait supposer que le contenu de la FAT est :

```
ABCDEF
```

eh bien non, ce sera :

```
BCFADE
```

Pourquoi ? Cela semble manifestement très curieux (pour nous) de stocker ABCDEF en BCFADE mais ça ne l'est pas pour l'ordinateur qui aura beaucoup plus vite fait d'extraire un des deux éléments s'ils sont présentés de la sorte. Les éléments d'une FAT à 12 bits seront donc regroupés 2 par 2 (2*1 byte et 1/2 =3 bytes) avec la représentation indiquée. Voici la méthode de consultation d'un élément de la FAT.

On multiplie le numéro de cluster donné (l'élément à lire) par 3 et puis on divise par 2, le résultat est utilisé comme un déplacement dans la FAT, on lit alors 2 bytes. Dans ces deux bytes, nous avons 4 signes hexa dont seulement 3 nous intéressent, il faudra donc éliminer soit le premier signe soit le dernier, cela dépend si le numéro de cluster est pair ou impair : rappelons qu'un nombre ABCD est toujours représenté en machine sur 2 bytes avec la forme CDAB (inversion des bytes). Dans notre exemple, nous obtenons donc les nombres sur 2 bytes :

BCFA (cluster Pair) ou FADE (cluster impair),

Ces bytes représentent les nombres FABC et DEFA (en inversant les bytes). On voit que le premier contient ABC mais que le premier signe hexa est de trop tandis que le second contient DEF mais le dernier signe hexa est de trop, on va donc éliminer le signe superflu en utilisant le masque approprié et on aura le contenu de l'élément.

Imaginons que ces deux valeurs (ABC, DEF) soient respectivement les contenus des éléments 4 et 5 de la FAT :

```
Contenu           : BC   FA   DE
                     ^    ^    ^
Offset en bytes   : 6    7    8
```

• On recherche le contenu de l'élément 4 :

- 4*3/2 = 6, on lit les bytes 6 et 7 --> BCFA
- Nous inversons les bytes --> FABC
- Comme le numéro de cluster (4) est pair, on élimine le premier signe, ce qui donne 0ABC et le tour est joué.

• On recherche le contenu de l'élément 5 :

- 5*3/2 = 7.5.
- On tronque --> 7.
- On lit les bytes 7 et 8 --> FADE.
- On inverse les bytes --> DEFA.
- Comme le numéro de cluster (5) est impair, on élimine le dernier signe, ce qui donne DEF0.
- On cadre le nombre à droite -->0DEF et le tour est joué.

Exemple de programmation

Continuons l'élaboration de notre programme, voici une fonction qui retourne le contenu d'un élément de la FAT, cette fonction teste une variable calculée par la procédure de lecture de la zone de démarrage (cf. pages précédentes) qui indique s'il s'agit d'une FAT 12 ou 16 bits (on a pu le déterminer en comparant le nombre de clusters avec la valeur 4086, cf. ci-dessus) :

```
Function
ElSuivantDeFat(cluster:integer):Integer;
{-------------------------------------}
var
 el,temp:integer;
begin
 el:=0;
 If Params.BitsdeFat=16 Then
  move(fat[cluster shl 1],el,2)
 else
 begin
  Temp := 3*Cluster;
  Move(FAT[Temp SHR 1], el, 2);{SHR = /2}
  IF Odd(Cluster) THEN {Cluster impair ?}
  begin  {oui}
   el := el AND $FFF0;
   {on supprime le dernier signe hexa }
   el := el shr 4;{on "recadre" l'élément}
  end
  Else  {cluster Pair}
   el := el AND $0FFF;
  {on supprime le premier signe hexa}
 End;

 ElSuivantDeFat:=el;
end;
```

Les répertoires et les fichiers

Le répertoire principal est construit au formatage et possède **une taille fixe**, il occupe un certain nombre de secteurs contigus sur le disque. Le numéro du premier et le nombre de secteurs peuvent être lus dans la zone de démarrage. Les sous-répertoires, quant à eux, sont des fichiers et n'ont donc pas de taille limite; on trouvera dans le répertoire le premier cluster de ce "fichier" et il faudra consulter la FAT pour connaître les autres. Un répertoire (principal ou sous-répertoire) est constitué d'éléments de 32 bytes qui répondent au format suivant :

Bytes	Signification
0..7	Nom du fichier, le premier byte du nom du fichier indique le status, les valeurs possibles sont :
	00 — Il n'y a jamais eu de nom de fichier dans cet élément du répertoire.
	E5 — Le fichier a été effacé.
	2E — Cet élément est un sous-répertoire («.») et la valeur pointe dans le répertoire courant. Si le second byte est aussi 2E («..»), il s'agit alors d'un pointeur vers le répertoire contenant celui-ci (répertoire parent).
	Autre — indique que c'est la première lettre d'un nom de fichier.

```
8...10  Extension du fichier.
11      Attribut, les valeurs possibles sont:

          00        Fichier normal.
          01        Fichier à lecture uniquement
                    (read only).
          02        Fichier caché.
          04        Fichier système.
          08        Etiquette de volume
                    (unique).
          10        Sous-répertoire.
          20        Fichier à "backuper".

12..21  Réservés.
22..23  Heure (hhhhhmmmmmmmssssss, les lettres
        représentent les bits utilisés pour
        heure, min., sec.).
24..25  Date (aaaaaaammmmjjjjj).
26..27  Premier cluster du fichier (--> FAT).
28..31  Taille.
```

Note: Dans le cas d'un fichier effacé, le premier caractère de son nom est mis à '?' mais le numéro de cluster n'est pas effacé. Par contre, tous les éléments de la FAT indiquant les clusters qu'occupait ce fichier sont remis à 0 (indiquant que le cluster est de nouveau libre).

Maintenant que nous savons que les fichiers sont alloués par cluster et que les informations de la FAT sont aussi des clusters, nous devons disposer de formules de conversion cluster--> secteur et secteur--> cluster pour être en mesure d'utiliser notre interruption 25h :

- Cluster --> Secteur :

```
With Params Do
  Secteur:=(Cluster-2)*SectsPCluster+PremData;
```

Cluster-2 car les deux premiers éléments de la FAT ne sont pas pas utilisés (réservés).

- Secteur --> Cluster :

```
With Params Do
 Cluster:=Trunc((secteur-
   premdata)/SectsPCluster+2);
```

Nous pouvons donc connaître le contenu du répertoire principal en lisant les secteurs qui le composent (interruption **25h**) et en extrayant 16 éléments de 32 bits par secteur (16*32 = 512 bytes).

Pour lire le contenu d'un fichier du répertoire principal :

1. Nous utilisons le numéro du premier cluster (contenu dans les 32 bits).

2. Nous convertissons ce numéro de cluster en numéro de secteur et nous utilisons l'interruption **25h** pour lire tous les secteurs (SectsPCluster) de ce cluster.

3. Nous lisons l'élément (correspondant au cluster) de la FAT avec la fonction "ElSuivantDeFat" (cf. pages précédentes), on obtient un autre numéro de cluster (le suivant pour ce fichier).

4. Si la valeur lue est inférieure à **FFF8h** nous recommencons en (2), sinon c'est fini.

Pour consulter les fichiers d'un sous-répertoire, c'est la même chose : on décomposera à nouveau chaque secteur en éléments de 32 bits, etc. Et si dans ce sous-répertoire on trouve un "sous-sous-répertoire" ? Eh bien on recommence une fois encore : on utilise le premier cluster pour consulter la FAT,... et ainsi de suite.

Exemple de programmation

Voyons maintenant comment programmer cela mais étape par étape. D'abord le répertoire principal, puis la consultation de la FAT (calcul du nombre de clusters d'un fichier), puis les sous-répertoires...

Voici une procédure de lecture du répertoire principal, elle se borne, pour l'instant, à afficher le nom, la taille, le premier cluster et les attributs d'un fichier. Le numéro du premier secteur du répertoire principal et le nombre de secteurs ont été obtenus par lecture de la zone de démarrage, eh oui...

Notes :

1. Toutes les variables sont celles de la version finale, certaines seront donc expliquées plus bas.

2. Nous avons ajouté des repères (*..*) qui nous aiderons à insérer, au fur et à mesure, les parties manquantes.

```
Procedure LitRepertoire( Drive,
                         PremierSect,
                         NbrSect:Integer;
                         RootDir:Boolean;
                         masque:str12);
{---------------------------------------}
{Les paramètres d'entrée de la procédure }
{sont les suivants:                      }
{                                        }
{- Numéro du disque,                     }
{- Numéro du premier secteur du          }
{  répertoire                            }
{- Nombre de secteurs du répertoire      }
{- Un flag (TRUE ou FALSE) indiquant s'il }
{  s'agit du répertoire principal ou non, }
{- Un masque éventuel                    }

Const
  {ces constantes représentent les valeurs   }
  {des attributs, cf. Chap. Les répertoires  }}
 Normal = 0;
 Lecture = 1;
 Cache = 2;
 Systeme = 4;
 Volume = 8;
 Repertoire =16;

Type

  {détail d'un des éléments de 32 bits du     }
  {répertoire    }
 EntreeRepert=Record
  NomFichier    :array[1..8] Of Char;
  ExtFichier    :array[1..3] Of Char;
  AttFichier    :Byte; {attribut}
  Rien          :array[1..10] of char;
  heure         :integer;
  date          :integer;
  PremCluster   :integer;
  tb1           :Char; {4 caract (code ASCII)  }
                       {pour taille            }
  tb2           :Char;
  th1           :char;
  th2           :char;
 end;
```

```
var
  tempreal :real;
  FinList : Boolean;
  erreur,i,j,k,l : Integer;

  {Numéro de secteur à lire}
  Secteur : Integer;

  {buffer pour int25}
  Buffer : array[0..TailleSect] of char;


  {contient les 32 bytes d'un él. de la dir}
  UnElement : EntreeRepert;

  sdc : byte; {Num. de secteur dans un cluster}
  Choix : str12;{utilisé pour les masques}


(* 0 *)

begin {LitRepertoire----------------}

  (* 1 * (repère pour la suite))

  {affiche un titre...}
  writeln;
  writeln(Indent,'FICHIER ',' ','EXT',' ',
                 'PremC',' ', ' Taille',' ',
                 ' date heure ', 'NBClu',
                 ' ','Attribut');

  writeln(Indent,'========',' ','===',' ',
                 '=====',' ', '=======',' ',
                 ' ==== ===== ',
                 '=====',' ', '========');

   {"Secteur" contient le secteur à lire et  }
   {analyser ensuite                         }
  Secteur:=PremierSect;
  Sdc:=0;
  i:=0;
  while (i<NbrSect) Do
   {Parcours de tous les secteurs de la dir}
  begin
    {lit un secteur:}
```

```
  Int25(drive,Secteur,1,Buffer,Erreur);
  If Erreur<>0 Then
  begin
   writeln('erreur de lecture du répertoire');
   halt;
  end;

  j:=0;
  While (j<512) Do
   {Découpe le secteur en unités de 32 bytes }
   {Pour décoder les informations des        }
   {fichiers                                 }
  begin
   Move(Buffer[j],UnElement,32);
   With UnElement Do
   begin

(* 2 *)

    If nomfichier[1]<>chr(0) then
     {=0 alors entrée du répertoire     }
     {encore jamais utilisée.....       }
    begin
(* 3 *)

(* 4 *)
    {laisser le "begin" pour la suite *)
     begin
      write (Indent);
      write(Nomfichier,' ',extfichier,' ',
            PremCluster:5,' ',
            Char4EnReal(tb1,tb2,th1,th2):7:0,
            ' ',' date heure ');

      write(' ');

(* 5 *)

       {décode les attributs et affiche...}
       {supprime les bits 8,7,6 2.}
       AttFichier:=AttFichier And 31;
       {31=00011111}
       If attfichier=0 then
        write('Archive ')
       else
       begin
```

```
            if (attfichier and lecture)<>0 then
              write('Read Only ');
            if (attfichier and cache)<>0 then
              write('Hidden ');
            if (attfichier and volume)<>0 then
              write('Volume ');
            if (attfichier and systeme)<>0 then
              write('Système ');
            if (attfichier and repertoire)<>0 then
              write('<Répertoire> ');
           end;{if attfichier... else}

          If ord(nomfichier[1])=$E5 then
           {code pour fichier effacé}
          write(' Effacé')

          writeln;
         end;{}
        end;{if <>chr(0)...}

(*  6  *)

     end;{with}
     j:=j+32;
    end; {while j}

(*  7  *)

    Secteur:=Secteur+1; {secteur suivant}
    i:=i+1; {compte le nombre de secteurs lus}
   end; {while i}

end;{LitRépertoire}
```

Et dans notre programme des pages précédentes, on rajoute la ligne suivante juste après "LitFat..." (avant le "end" du "With") :

```
LitRepertoire( Disque,PremRoot,SectsPRoot,
               True,'*.*');
```

Si nous souhaitons que notre programme calcule le nombre de clusters occupés par un fichier (ou sous-répertoire), nous allons les compter en suivant la liste dans la FAT, dans ce but nous utilisons la fonction ElSuivantDeFat (cf. pages précédentes).

Nous devons ajouter:

1. à l'indice `(* 3 *)` le code suivant:

```
k:=0;
 {k compte le nombre de clusters}
l:=premcluster;

repeat
  {consulte la fat et calcule le nombre }
  {de clusters d'un fichier ou répertoire }
 l:=elsuivantdefat(l);
 If l=0 Then
  Finlist:= True
  {cas d'un fichier effacé }
  {cf. note ci-dessus }
 Else
 begin
  k:=k+1;
  If Params.BitsdeFat=16 Then
   {$FFF8 est un nombre trop grand, }
   {et donc                         }
   finlist:=((l SHR 4) = $FFF)
            AND ((l AND $F) >= 8)
  else
   finlist:=l>=$FF8;
 end;
until (finlist);
{maintenant k=Nombre de clusters}
```

2. à l'indice `(* 5 *)`, remplacer:

```
write(' ');
```

par :

```
write(k:5,' ');{nombre de clusters}
```

Si nous voulons maintenant que les sous-répertoires soient aussi consultés, nous devrons parcourir la FAT pour connaître les clusters qui contiennent le répertoire. Pour chaque cluster nous devrons lire ses secteurs et les analyser. Pour réaliser cela, nous utiliserons la **récursivité** : dès qu'un répertoire est rencontré, on appelle directement la procédure "litRepertoire" (dans laquelle on est) en lui transmettant deux paramètres :

1. Le premier secteur du répertoire (que nous pouvons calculer avec le numéro du cluster, qui est connu puisqu'il est dans les 32 bits).

2. Le nombre de secteurs = Nombre de clusters * Nombre de secteurs par cluster, le nombre de clusters ayant déjà été calculé.

Ce n'est pas encore fini... Les secteurs d'un sous-répertoire ne sont pas nécessairement contigus, il faut travailler avec les clusters lus dans la FAT et pour chaque cluster, il faudra lire SectsPCluster Secteurs.

Au repère `(* 7 *)`, on remplace :

```
Secteur:=Secteur+1;
```

par :

```
If RootDir then
 {en root dir, les secteurs sont }
 {contigus d'office              }
  Secteur:=Secteur+1
else
```

```
{dans un ss-répertoire, il faut consulter  }
{la FAT, un ss-répertoire est traité comme }
{un fichier   }
with params do
begin
 Sdc:=sdc+1;{Num. du sect. dans le cluster}
 if sdc>(SectsPCluster-1) Then
  {on a lu le dernier sect. du cluster,     }
  {on change de cluster                     }
 begin
  {On traduit secteur en cluster pour       }
  {consult. de la FAT      }
  k:=trunc((secteur-premdata)/SectsPCluster+2);
  l:=elsuivantdefat(k);
   {On traduit cluster en secteur}
  secteur:=(l-2)*SectsPCluster+premdata;
  sdc:=0;     {réinit. à 0, c'est le    }
              {premier sect. du cluster     }
 end
 else
   {on n'a pas lu le dernier secteur du cluster}
   {dans un cluster, les secteurs sont contigus}
  Secteur:=Secteur+1;
end;{else with param}
```

On appellera donc cette procédure avec la variable RootDir à FAUX car ce n'est pas le répertoire principal, et cela influencera la suite.

Insérer le code suivant au repère `(* 6 *)` :

```
With params Do
  {on consulte le ss-répertoire}
 if (attfichier=Repertoire)
    and (Nomfichier<>'. ')
    and (Nomfichier<>'.. ')
 then
 begin
  Indent:=Indent+' ** '; {augmente          }
                         {l'indentation     }
  l:=(Premcluster-2)*4+premdata;
   {convertit prem. cluster numéro de   }
   {secteur      }
```

```
    k:=k*SectsPCluster;
     {convertit nombre de clusters en nombre   }
     {de secteurs }
    LitRepertoire(2,1,k,NoRoot,masque);
     {appel récursif }
    Indent:=copy(Indent,1,length(Indent)-4);
     {diminue l'indentation }
  end;
```

Finalement, pour que notre programme supporte aussi les masques de recherche, nous devons ajouter :

1. Au repère `(* 0 *)`, la procédure suivante :

```
Procedure Mask(Var Masque:str12),
{------------------------------}

 {Ajuste un masque comme suit:      }
 {                                  }
 { 'P*.EX?' -->   'P???????.EX?'    }
 { '*.*'    -->   '????????.???'    }
 { '*.'     -->   '????????.   '    }
 { '.*'     -->   '????????.???'    }
 { 'PA*.*'  -->   'PA??????.???'    }
 { '*.A*    -->   '????????.A??'    }

Type
 str3=string[3];
 str8=string[8];

Var
 Etoile,point,k:byte;
 Nom:str8;
 Ext:Str3;

Begin {mask}
  {met tout en majuscules}
 For k:=1 to Length(masque) do
  masque[k]:=upcase(masque[k]);

 Nom:='????????';
 Ext:='???';
 Point:=Pos('.',Masque);
```

```
{Analyse l'extension:}
If Point<>0 Then
begin
 Ext:=Copy(Masque, point+1,length(masque)-
      Point);

   {Pour '*.' --> '*.   '}
 If Ext='' Then Ext:='   ';

  {complète si moins de 3 caractères}
 Ext:=Ext+'???';

  {S'il y a une étoile...}
 Etoile;:Pos('*',Ext);
 If Etoile<>0 then
   {Tous des '?' à partir de l'étoile}
  For k:=etoile to 3 do
   Ext[k]:='?';

  {Point=dernière position du nom}
 Point:=Point-1;
end
else
 point:=Length(Masque);{simule le point}

 {analyse le nom:}
Nom:=copy(masque,1,Point)+Nom;

Etoile:=pos('*',nom);
If Etoile<>0 Then
 For k:=Etoile to 8 do
  Nom[k]:='?';

{Prépare le masque que l'on renvoie modifié:}
Move(Nom[1],masque[1],8);
Masque[9]:='.';
Move(ext[1],Masque[10],3);
Masque[0]:=chr(12);

End;{mask}
```

2. Au repère (* 1 *), le code suivant :

```
If RootDir Then  {On n'analyse le masque  }
                 {qu'une seule fois       }
 Mask(Masque);
```

3. Au repère (* 2 *) :

```
Choix:=NomFichier+'.'+ExtFichier;
For k:=1 to 12 Do
 If Masque[k]='?' then
  Choix[k]:='?';
```

4. Au repère (* 4 *) :

```
{Affiche d'office le nom du répertoire même }
{s'il ne correspond pas au masque pour      }
{qu'il y ait un en-tête à la liste des      }
{fichiers.                                  }

If (choix=masque) or (attFichier=repertoire)
then
```

Notons qu'il est **impératif** de ne pas modifier les variables déclarées **locales** car cela pourrait endommager la récursivité.

7. Récupération des fichiers effacés

Il faut savoir que lorsqu'on crée un nouveau fichier, MS-DOS lui alloue, si possible, des clusters contigus. La seule technique possible de récupération de fichiers effacés est la suivante :

- Demander à l'utilisateur de sélectionner le fichier à récupérer ainsi que la première lettre de son nom puisque celle-ci est effacée.

- Diviser la taille du fichier (contenue dans l'entrée du répertoire) par le nombre de bytes par cluster pour connaître le nombre de clusters qu'occupait le fichier, cela donne un nombre *x*.

- Consulter la FAT à partir du premier cluster qu'occupait ce fichier (cette valeur est toujours dans l'entrée répertoire du fichier effacé) et vérifier qu'il y a *x* éléments qui se suivent qui contiennent 0 (cluster libre).

- Si ce n'est pas le cas, le fichier n'est pas récupérable; l'espace qu'il occupait a été entièrement ou partiellement attribué à un autre fichier.

- Si c'est le cas, la récupération du fichier est probable mais pas certaine (nous verrons ci-dessous pourquoi). Il faut alors :

  - Recréer la liste en modifiant l'élément (*i*) pour qu'il contienne la valeur *i+1* et modi-

fier le dernier élément pour qu'il contienne FFFh (marque de fin de liste).

- Restaurer la première lettre du nom du fichier dans l'entrée du répertoire.

Il faut savoir que tant qu'il y a de la place sur le disque, MS-DOS ne récupérera jamais l'espace libéré par des fichiers effacés : il allouera toujours de l'espace disque pas encore utilisé. Ainsi, tant que c'est le cas, la récupération de fichiers effacés est garantie. Dès que tout le disque a été occupé au moins une fois, les fichiers comporteront peut-être des trous : les clusters de certains fichiers ne seront plus contigus. Dans ce cas, la chance de récupération d'un fichier est nulle puisqu'il est impossible de connaître la position des ensembles de clusters autres que le premier. Pour vérifier cette affirmation, il suffit de prendre des programmes de récupération et d'appliquer la procédure suivante :

- Remplir une disquette avec des fichiers de taille normale (+/- 20K); il doit rester un minimum d'espace disponible.

- Sélectionner un fichier de +/- 80K n'existant pas sur la disquette.

- Libérer de la place sur la disquette pour ce fichier en effacant des fichiers de la taille la plus petite possible.

- Copier ce fichier sur la disquette.

- Effacer toute la disquette.

- Essayer de récupérer le fichier.

Le programme de récupération dira certainement qu'il est possible de récupérer le fichier (car il y a x éléments contigus de la FAT à 0), mais le contenu du fichier ne sera pas correct. Notons aussi que, dans le cas où il n'y a pas x éléments de la FAT contigus qui sont libres, certains programmes de récupération continuent à analyser la FAT séquentiellement pour trouver des clusters libres (élément de la fat = 0) et les ajoutent au fichier : cette méthode donnera très rarement de bons résultats. Une meilleure solution consiste à afficher le contenu à l'écran chaque fois qu'on trouve un cluster libre et de demander à l'utilisateur si ces données font partie du fichier à récupérer.

Exemple de programmation

Nous n'allons pas ici écrire un programme de récupération des fichiers effacés mais juste donner deux procédures de manipulation de la FAT. Ceux qui souhaitent écrire ce programme trouveront les autres procédures nécessaires (ainsi que les variables) dans le programme "bootFatDir".

```
Procedure ValeurDansFat(Cluster,Valeur:Integer);

  {Cette procédure écrit "valeur" dans    }
  {l'élément de la fat correspondant au   }
  {cluster "cluster"                      }
 Var
  el,temp:integer;

 begin
  If Param.BitsDeFat=16 Then
   move(valeur,fat[cluster shl 1],2)
  else
   begin
    If valeur>$0FFF then
    begin
     Writeln('Erreur, valeur pas sur 12 bits);
     exit;
    end;
    Temp:=3*Cluster;
    Move(Fat[Temp Shr 1],el,2);
    If Odd(cluster) then {cluster impair?}
    begin
     el:=el and $000F;
     valeur:=valeur shl 4;
     el:=el or valeur;
     Move(el,fat[Temp Shr 1],2);
    end
    else {le cluster est pair}
    begin
     el:=el and $f000;
     el:=el or valeur;
     move(el,fat[temp shr 1],2);
    end;
```

```
  end;{else if bitsdeFat}
end;{ValeurDansFat}
```

La procédure suivante réécrit la FAT que nous avons modifiée (avec la procédure ci-dessus) sur le disque. Nous pouvons modifier la procédure **Int25** pour créer **Int26** (écriture de secteurs logiques) en changeant simplement $CD/$25 en $CD/$26 (au milieu de la procédure).

```
Procedure
FatToDisk(Drive,premierSect,NbrSect:integer);

 var
 erreur,i,j:integer;
 secteur:integer;
 buffer:array[1..Taillsect] of char;

 begin

  Secteur:=premiersect;
  i:=0;
  j:=0;
  While (i<NbrSect) Do
  begin
   Move(fat[j],buffer,512);
   j:=j+512;

   Int26(drive,secteur,1,buffer,erreur);
   If erreur><0 then
   begin
    writeln('erreur d''écriture de la FAT');
    exit;
   end;

   secteur:=secteur+1;
   i:=i+1;
  end;{while i<nbrsect}

end;{fat to disk}
```

8. Autres applications

Il existe évidemment une multitude d'applications que l'on peut programmer dès que l'on maîtrise les techniques que nous venons de découvrir. Parmi celles que l'on pourra réaliser avec un minimum de modifications dans le programme "BootFatDir", on notera :

- Un programme qui recherche dans quels répertoires se trouve un fichier donné. Pour cela, il faudra simplement spécifier le nom du fichier à rechercher comme masque de la procédure "LitRepertoire" et évidemment modifier quelque peu la présentation du programme "BootFatDir".

- Un programme qui recherche et affiche toutes les occurrences d'un même fichier sur le disque. Ce programme est similaire à l'exemple précédent.

- Un programme qui calcule l'efficacité de stockage des fichiers et du disque en général :

  - Il faudra ajouter quelques lignes qui vérifient que les clusters sont contigus (dans la boucle "repeat" qui calcule le nombre de clusters du fichier). On pourra ainsi compter le nombre de "trous" dans l'espace disque occupé par un fichier.

  - On pourra déterminer qu'un fichier est dans de mauvaises conditions d'accès s'il possède

plus de *x* trous et calculer ainsi un pourcentage d'efficacité.

- Finalement, on calculera un pourcentage d'efficacité du disque (en prenant une moyenne des pourcentages des fichiers) qui déterminera si celui-ci doit être réorganisé.

Pour réorganiser un disque, il suffira d'en faire un backup complet (FILE backup et pas IMAGE backup avec un tape streamer), de le reformater et, enfin, de restaurer le backup. Une nouvelle exécution du programme devrait ainsi afficher **une efficacité de 100%** pour TOUS les fichiers.

- Le programme CHKDSK de MS-DOS analyse la FAT et recherche les éléments qui ne font pas partie d'une liste ou les listes qui n'ont pas de fin (pas de marque FFFh). L'option '/F' de CHKDSK, qui propose de convertir les clusters perdus en fichiers, crée des fichiers dans le répertoire principal et ajuste le premier cluster pour qu'il pointe vers un des éléments "perdus".

9. Récapitulation

☞ **Un secteur** est l'unité de base d'accès à un disque, il contient 512 bytes.

☞ **Un disque est divisé en pistes**, chaque piste contient 8 ou 9 secteurs.

☞ L'ensemble des pistes numéro "x" est appelé **cylindre.**

☞ Il y a deux numérotations possibles des secteurs :
- La numérotation **logique** (interruptions 25h et 26h).
- La numérotation **physique** (interruption 13h).

☞ Le secteur physique 0 contient des informations sur les **partitions.**

☞ A chaque partition est associée une unité de disque.

☞ L'unité d'allocation d'espace disque à un fichier est appelée "**cluster**".

☞ Un disque est découpé en plusieurs éléments :
- Le "boot record".
- La FAT.
- Le répertoire principal.
- Les fichiers.

☞ Tous les clusters d'un fichier sont répertoriés dans la FAT sous forme de liste.

Chapitre 7

La gestion de la mémoire et les programmes

1. Le processus "parent appelle fils"

Le système d'exploitation MS-DOS est un système monotâche, c'est à dire qu'à l'opposé de systèmes comme UNIX il ne peut exécuter qu'un programme à la fois. Cela ne signifie pas qu'il ne peut y avoir qu'un seul programme en mémoire à un moment donné. Un programme pourra en appeler un autre, mais dans ce cas, le second programme aura alors le contrôle complet de la machine pendant tout le temps où il s'exécute : le contrôle ne sera rendu au premier qu'au moment où le second se termine.

C'est ce schéma qui est à la base du fonctionnement du MS-DOS : le programme COMMAND.COM est chargé en mémoire au démarrage de l'ordinateur, ce programme ne se terminera JAMAIS (c'est COMMAND.COM qui affiche "A:>", "C:DOS>",...)! Ainsi, lorsque nous frappons une commande au clavier, COMMAND.COM déterminera s'il s'agit d'une commande interne (dir, copy,...) ou d'une commande externe (chkdsk, format, exécution d'un logiciel de traitement de texte, de comptabilité, etc.). Dans le second cas, la commande sera traitée par un programme appelé par COMMAND.COM; ce dernier ne reprendra la main que lorsque le programme appelé prendra fin et on verra alors réapparaître le "prompt" (A:> ou C:>) du DOS. Le programme qui est exécuté sur demande de COMMAND.COM peut aussi demander l'exécution d'un autre programme (et ainsi de suite), il peut même demander l'exécution de COM-

MAND.COM (c'est la fonction "Accès DOS" disponible dans certains programmes). Dans ce dernier cas, une seconde version de COMMAND.COM sera chargée en mémoire et exécutée.

- Le programme qui en appelle un autre se nomme le "**parent**" (parent process).

- Le programme qui est appelé se nomme "**fils**" (child process).

MS-DOS propose toute une série de fonctions permettant à un programme d'exploiter ces possibilités. Cependant, avant de les analyser, nous devons comprendre la structure de la mémoire, les zones réservées aux programmes, les techniques utilisées, etc.

2. Structure de la mémoire

La zone mémoire s'étendant jusqu'à 640 Ko est réservée à MS-DOS et aux programmes (traitement de texte, comptabilité,...). Elle est divisée en plusieurs blocs dont la taille, exprimée en paragraphes, peut en théorie atteindre presque 1MB (un paragraphe=16 bytes). Chacun d'eux est précédé d'une structure de 16 bytes appelée "**Memory Control Bloc**" (MCB) qui contient des informations sur la taille et le status du bloc de mémoire. Dans les versions MS-DOS 2.xx et supérieures, MS-DOS maintient une liste des blocs de mémoire.

Note : La partie basse de la mémoire (celle qui contient le vecteur d'interruption, les informations du BIOS et du DOS, les divers pilotes) n'est pas répertoriée dans la chaîne des blocs.

Un MCB est directement suivi du bloc mémoire qu'il contrôle, les éléments de la chaîne sont en mémoire les uns derrière les autres, on a donc le schéma suivant :

```
paragraphe  contenu
----------  --------------------------
0000        Mémoire Basse

            Premier MCB (16 bytes).
            Premier bloc mémoire (taille va-
            riable).
            MCB suivant (16 bytes).
            Bloc mémoire suivant (taille va-
            riable).
            ...
            Dernier MCB (16 bytes).
            Dernier bloc mémoire (taille va-
            riable).
nnnn        Fin de la mémoire.
```

Chaque fois qu'une demande de mémoire est faite par un programme (un programme peut faire plusieurs demandes), MS-DOS inspecte la chaîne des blocs afin d'en trouver un dont la taille est suffisante pour satisfaire à la demande. Un programme ne peut utiliser que la mémoire qui lui a été attribuée : s'il tente d'écrire des informations en dehors de la zone prévue, il risque d'endommager un des MCB. MS-DOS réagira alors en envoyant le message "***memory allocation error***" et il arrêtera le système immédiatement !!

L'adresse du premier MCB peut bien entendu varier en fonction de la version du DOS, des drivers installés (via config.sys), des buffers et autres zones réservées pour des usages particuliers. Heureusement, MS-DOS dispose d'une fonction (non documentée) qui permet d'obtenir l'adresse de segment du premier MCB.

La fonction **52h** de l'interruption **21h** renvoie l'adresse du premier MCB.

Cette fonction nous renvoie deux valeurs dans les registres ES et BX du microprocesseur, l'adresse de segment du premier MCB est :

```
ES:[BX-2]
```

On pourra donc trouver cette adresse au moyen de quelques instructions :

```
mov ah,52H              ; fonction 52h
int 21H                 ; appel à MS-DOS
mov ax,es:[bx-2]        ; AX <-- adr du
                        ; segment du MCB
```

Structure d'un MCB

Un MCB est un bloc de 16 bytes qui contient les informations suivantes :

Offset	Longueur	Contenu
0	1 Byte	**'M'**:indique que ce n'est pas le dernier bloc de la chaîne **'Z'**:indique que c'est le dernier bloc de la chaîne **Autre**:indique que les blocs mémoire ont été écrasés et MS-DOS ne va pas tarder à nous envoyer le message "memory allocation error" et rapidement arrêter le système...
1	2 bytes	Contient l'adresse de segment du programme à qui le bloc de mémoire appartient; plus exactement, c'est l'adresse du PSP de ce programme (cf. chap. suivant).
3	2 bytes	Contient la taille du bloc exprimée en paragraphes.
5-15		Reservés

L'adresse du segment du premier bloc et les informations du bloc sont suffisantes pour pouvoir parcourir la chaîne complète.

Exemple de programmation

Voici, en Turbo Pascal, un exemple pour lire et interpréter la chaîne des MCB :

```
Program AfficheBlocs;

type

 Registres = record
  case Integer of
    1: (ax,bx,cx,dx,bp,si,di,ds,es,flags:Integer);
    2: (al,ah,bl,bh,cl,ch,dl,dh : Byte);
 end;

Var
 reg:registres;
 segmcb,proprietaire,taille,adresse:integer;

begin

  {Appel à MS-DOS avec la fonction 52h}
 Reg.ah:=$52;
 MsDos(reg);

  {SEGFCB <-- adresse}
 SegMcb:=MemW[Reg.es:Reg.bx-2];

  {boucle tant que le premier byte du MCB     }
  {='M', ce qui signale que le dernier bloc de la}
  {chaîne n'est pas atteint       }
While (Mem[SegMcb:0]=Ord('M') Do
begin
  {afficher le contenu du MCB ou tout   }
  {autre traitement                     }
 Proprietaire:=MemW[Mcb:1];
 taille:=MemW[Mcb:3];
 writeln('le bloc est.',SegMcb,
         ' ',proprietaire,
         ' ',taille);
```

```
    {calcule l'adresse de segment du MCB      }
    {suivant qui se trouve après le bloc      }
    {contrôlé par l'actuel MCB    }
   SegMcb := SegMcb + taille + 1
  end;{while}

  If (Mem[SegMcb:0]=Ord('Z') Then
  begin
    {afficher le contenu du MCB ou tout autre }
    {traitement  }
   Proprietaire:=MemW[Mcb:1];
   taille:=MemW[Mcb:3];
   writeln('le dernier bloc est.',SegMcb,
           ' ',proprietaire,
           ' ',taille);
  end
  Else
    {Ni 'M' ni 'Z', c'est inquiétant....}
   writeln('Memory allocation error');

end.{programme}
```

Grâce aux données contenues dans le MCB, il est aussi possible de déterminer les informations suivantes :

- Le nom du programme à qui le bloc mémoire appartient (cf. le chapitre sur la structure d'un programme en mémoire, le PSP et l'environnement).

- Le nom du parent (cf. le chapitre sur le processus parent-fils).

- Les paramètres du programme au chargement (cf. le chapitre sur la structure d'un programme en mémoire : le PSP et l'environnement).

- Le nombre de blocs mémoire qu'un programme possède (cf. le chapitre sur la structure d'un programme en mémoire : le PSP et l'environnement).

- La taille totale de la mémoire allouée à un programme (calcul de la somme des tailles des blocs que ce programme possède).

- Les interruptions qu'il a modifiées : quand on connaît l'adresse et la taille des blocs alloués à un programme, on peut facilement vérifier si l'adresse de l'interruption pointe dans une zone qui appartient au programme ou non et, dès lors, en déduire si ce programme a modifié l'interruption. Si plusieurs programmes ont modifié successivement la même interruption (programmes *résidents*), il devient relativement difficile de suivre la "piste".

On pourra donc améliorer légèrement le petit programme (ci-dessus) pour qu'il affiche, par exemple, les programmes *résidents*; ce qui donnerait :

Liste des programmes *résidents* :

```
PSP blocs bytes     Prog   Paramètres     Interruptions
                                            modifiées
--- ----- -----  --------  ----------  -----------------

0008    1   24384    config
0FB1    3   3584     command                      2E
109C    2   5584     KEYB      FR,437,c:\dos\ke   09 2F FF
11FB    2   2512     DOSEDIT   N/A                21
129E    3   346672   FW                           10 1C 63 64
6744    3   3648     MSPEC=/COMMAND               22 24
682B    2   228688   Libre
```

On peut, par exemple, voir à la ligne 5 que j'ai exécuté ce programme en étant en Accès DOS de Framework (FW), et que ce dernier occupe 346672 bytes de la mémoire répartis sur trois blocs. De plus, FW a modifié les interruptions 10h, 1Ch, 63h et 64h.

3. Stratégies d'allocation

Nous savons que MS-DOS parcourt la chaîne des blocs lorsqu'un programme fait une demande de mémoire. Mais que se passe-t-il s'il y a plusieurs possibilités ? En effet, on peut envisager qu'après un certain temps de travail avec l'ordinateur, la mémoire soit découpée en une multitude de blocs, et que plusieurs d'entre eux pourraient être "candidats" pour l'attribution au programme ayant fait la requête. En fait, il existe trois stratégies d'allocation possibles :

1. On attribue le premier bloc de la chaîne suffisamment grand (first fit = choix du premier).

2. On attribue le plus petit bloc parmi ceux qui sont suffisamment grands pour satisfaire la requête (best fit = meilleur choix).

3. On attribue le dernier bloc de la chaîne suffisamment grand (last fit = choix du dernier).

.es tests ont montré que la première stratégie premier choix) est celle qui donne les meilleurs résultats; c'est donc la stratégie utilisée par défaut par 1S-DOS. Par défaut ? Oui, en fait, MS-DOS propose ne fonction qui permet à un programme de changer ı stratégie et de spécifier qu'il souhaite, par exemple, ue MS-DOS utilise la stratégie du meilleur choix.

La Fonction **58h** de l'interruption **21h** modifie la stratégie d'allocation mémoire (version MS-DOS **3.xx** uniquement).

Notons aussi que, lorsqu'un bloc a été désigné (quelle que soit la stratégie) et que sa taille est supérieure à la quantité de mémoire demandée par le programme, MS-DOS découpera le bloc en deux parties : une pour satisfaire la demande — laquelle aura la taille exacte souhaitée — et une autre qui sera inscrite dans la chaîne comme n'appartenant à personne.

4. Stucture d'un programme en mémoire

Lorsqu'un programme est chargé en mémoire, MS-DOS lui attribue deux zones mémoires (blocs) distinctes :

- Une zone contenant le **PSP** (Program Segment Prefix), le programme, les données et le stack (la pile).

- Une zone contenant **l'environnement.**

Le PSP (Program Segment Prefix)

Le PSP est une zone de 256 bytes qui se trouve avant chaque programme en mémoire. Le PSP est installé au début (offset **0000h**) d'un segment de la mémoire appelé CODE SEGMENT. Ce PSP contient des informations nécessaires à l'exécution du programme. Il a été conçu, à l'origine, pour maintenir une compatibilité avec le système d'exploitation CPM et permettre d'adapter facilement au MS-DOS des programmes écrits pour CPM; certaines informations qu'il contient ne servent donc plus à rien (cf. Chap. Evolution du MS-DOS).

On trouvera notamment dans le PSP (offset exprimés à partir du début du PSP) les éléments ci-après :

- Les adresses des interruption suivantes:

```
20h  -   (offset = 0Eh)
23h  -   (offset = 0Ah)
24h  -   (offset = 12h)
```

  Pourquoi? Eh bien, dans le processus "parent appelle fils", il n'y a rien qui empêche le "fils" de modifier les adresses des routines de gestion des interruptions (situées dans les premiers 1024 bytes de la mémoire) pour réécrire certaines de ces routines. Il est nécessaire de sauvegarder les adresses de certaines routines "critiques" au moment du démarrage d'un programme : cette pré-

caution permet d'éviter qu'un programme "fils", qui a modifié ces adresses, oublie de les rétablir à leur valeur originale avant de rendre le contrôle au parent. Ces adresses modifiées pointeraient alors vers une zone qui appartenait au programme fils (qui est terminé lorsque le parent reprend le contrôle) et tout transfert à une des interruption en question provoquerait fort probablement un "plantage" de la machine puisqu'il y aurait transfert dans une zone mémoire n'appartenant pas au programme. Les adresses sauvegardées sont celles connues au moment où le programme fils est chargé en mémoire; elles seront rétablies quand le programme fils prendra fin.

- La taille de la mémoire disponible exprimée en paragraphes :

```
offset 02h= adresse du dernier segment
alloué au programme (sur 2 bytes).
```

- Une zone contenant des informations sur les "handles" utilisés par le programme (non documenté), offset **18h** (maximum 20 handles par programme).

- L'adresse du segment de l'**environnement** (offset 2Ch sur 2bytes); cet environnement permet au parent de passer des informations au programme fils (cf. ci-dessous).

- Deux FCB (offsets **5Ch** et **6Ch** respectivement, longueur 9 bytes). Les FCB (File Control Block) étaient utilisés dans les versions **1.xx** de MS-DOS

(cf. le chapitre 3 consacré à l'évolution de MS-DOS).

- La longueur du string de paramètres (offset 80h sur 1 byte).

- Le string de paramètres (offset 81h sur 127 bytes).

Le programme parent pourra passer certains paramètres au fils (noms de fichier, switch). On aura par exemple (COMMAND.COM est le parent, FORMAT ou DISKCOPY sont les fils) :

```
- FORMAT /s  où /s est un paramètre.
- DISKCOPY A: B: où A: et B: sont les
  paramètres.
- etc.
```

Ces paramètres représentent en fait — excepté le nom du programme lui-même — tout ce que l'utilisateur à frappé sur la ligne de commande avant de taper RETURN; l'ensemble de ces paramètres est appelé "**command tail**".

- DTA (Disk Transfert Area) par défaut (offset 80h sur 128 bytes).

Cette DTA est une zone mémoire qui est utilisée comme buffer de données (zone de transfert) lors d'opérations de lecture/écriture sur disque (n'est quasi plus utilisé dans les versions MS-DOS **2.xx** et supérieures, cf. le chapitre 3 sur l'évolution du MS-DOS).

Exemple de programmation

Avec l'adresse de segment du PSP d'un programme, on peut donc connaître toute une série d'informations utiles; voici un petit programme qui compte le nombre de "handles" utilisés par le programme:

```
Function
HandlesUtilises(psp : Integer) : Integer;
 {Cette fonction calcule le nombre de    }
 {handles utilisés par le programme,     }
 {les codes 0..2 sont pour les           }
 {périphériques standard, le code FFh    }
 {indique que le handle n'est  pas       }
 {utilisé                                }

var
 i,j: Integer;
 El: Byte;
begin
 i := 0;
 if (psp <> 8) then
   {Le PSP à l'adr 8 est réservé pour DOS}
  for j := 0 to 19 do
  begin
   El := Mem[psp:$18+j];
   if not(El in [0,1,2,$FF]) then
    i:=i+1;
  end;
 HandlesUtilises := i;
end {fonction} ;
```

L'environnement

L'environnement permet au "parent" de passer des informations au "fils" sous forme de chaînes de caractères terminées par le caractère dont le code est ASCII 0 (chaîne de caractère en format appelé ASCIIZ). Ces chaînes de caractères sont de la forme :

```
nom=variable
```

La fin de la chaîne de caractères est signalée par le caractère dont le code ascii est 0. L'adresse de segment de l'environnement (commence toujours sur une limite de paragraphe) se trouve dans le PSP (cf. pages précédentes); si cette adresse est nulle, le programme utilisera l'environnement du parent. L'environnement général est celui de COMMAND.COM (qui est le premier programme exécuté au démarrage de la machine) que l'on peut initialiser au moyen de commandes comme PATH, SHELL, PROMPT ou SET.

Voici un exemple d'environnement (ici, le '*' indique l'emplacement du caractère ASCII 0) :

```
COMSPEC=C:\COMMAND.COM*PATH=C:\TURBO;C:\DOS;
        C:\FW*PROMPT=$P$G**
```

Exemple de programmation

A partir de la version 3.0 de MS-DOS, on peut trouver le nom du programme après l'ascii 0 de fin d'environnement. Voici un exemple de procédure (qui pourra être ajoutée à notre programme de liste des blocs) qui renvoie le nom du programme dont on donne l'adresse du PSP :

Remarque: Après l'ASCII 0, il y a d'abord un mot (2 bytes) puis le nom du programme avec le chemin (ex:C:\TURBO\TDEBUG.COM) et le nom est terminé par un ASCII 0.

```
Type Str64=String[64];

Function NomProg(Psp:Integer):Str64;

{Cette fonction a comme paramètre           }
{l'adresse de segment du PSP. Elle ira      }
{chercher l'adresse de l'environnement      }
{dans ce PSP et ensuite l'environnement     }
{(qui a une taille max de 32767 bytes)      }
{sera analysé pour repérer #0#0 (2 ascii 0).}
{Dès que l'on a trouvé, on refait une       }
{boucle pour extraire le nom.               }

Var
 env,i,k:integer;
 First,Found:Boolean;
 St :str64;
 Seg1,ofs1:integer;
begin
 Env := MemW[psp:$2C];
  {adr de l'environnement}

 Seg1:=Env;
 Ofs1:=0;
```

```
  i:=0;
  Found:=false;
  First:=false;

    {chercher #0#0...:}
  While Not(found) and (i<32767) Do
    {taille max=32767}
  begin
   If Chr(Mem[Seg1:Ofs1+i])=#0 Then
     If First Then
       Found:=true
     Else
       First:=true
   Else
     First:=False;
   i:=i+1;
  end;{while}

  If Not Found Then
    {pas trouvé...}
   NomProg:='Inconnu'
  Else
    {extrait le nom}
    begin
     St:='';
     i:=i+2 ;
      {saute les 2 premiers bytes}
     Found:=false;
     k:=0;
     while not(found) do
     begin
      if chr(mem[Seg1:OFs1+i+k])=#0 then
        found:=true
      else
        st:=st+chr(mem[Seg1:OFs1+i+k]);
      k:=k+1;
     end;{while not(found)}
     NomProg:=St;
    end;{else}
end;{Nomprog}
```

5. La mémoire réservée aux programmes

En général, tout le reste de la mémoire est d'office alloué au programme en cours; cependant, dans certaines situations, on souhaite diminuer la mémoire et éventuellement l'augmenter à nouveau par la suite. MS-DOS propose un certain nombre de fonctions prévues à cet effet :

- Interruption **21h**, fonction **4Ah** (modification de la taille d'un bloc mémoire).
- Interruption **21h**, fonction **48h** (demande d'allocation d'un bloc mémoire).
- Interruption **21h**, fonction **49h** (libération d'un bloc mémoire).

- La fonction **4Ah** permettra à un programme de demander une modification de la taille d'un bloc qu'il possède déjà.

- Si un programme fait une demande d'allocation de mémoire et qu'il en reste assez, la fonction **48h** créera alors une nouvelle zone de la taille spécifiée et indiquera au programme l'adresse de cette zone. S'il ne reste plus assez de mémoire, MS-DOS enverra un code d'erreur ainsi que la taille maximum disponible (exprimée en paragraphes).

- La fonction **49h** sera utilisée pour libérer une zone de mémoire et la rendre à nouveau disponible pour utilisation future. Quand le programme prend fin, tous les blocs qui lui étaient alloués sont automatiquement libérés par MS-DOS.

Remarque : MS-DOS regroupera en un bloc tous les bloc contigus. De ceci il résulte que, si les blocs alloués ne sont pas libérés quand un programme n'en a plus usage, il est possible que la mémoire soit divisée en une multitude de petits blocs non contigus. La conséquence en sera qu'aucune demande future ne pourra plus être satisfaite.

Exemple de programmation

Voici maintenant une procédure — utilisant la fonction 4Ah de l'interruption 21h — pour modifier la taille d'un bloc mémoire. Cette procédure sera nécessaire à un programme que nous développerons dans les pages qui suivent.

```
Procedure AjusteTaille(paras : Integer);

 {Ajuste la mémoire du programme à PARAS    }
 {paragraphes, CSEG est une variable        }
 {interne à TURBO PASCAL qui contient       }
 {l'adresse du segment de programme(CS)     }

var
 regs : Registres;
begin
 with regs do
 begin
  Ah := $4A;
  Es := CSeg;
  Bx := paras;
  MsDos(regs);
  if (Flags and 1)<>0 then
   {teste le flag CARRY}
  begin
   Case Ax of
     $07:writeln(^G,'Erreur d'allocation
                 de mémoire');
     $08:writeln(^G,'mémoire insuffisante');
     $09:writeln(^G,'Cseg incorrect...');
   end;
  end;
 end;
end;{AjusteTaille}
```

6. Exécution d'un programme "fils"

MS-DOS dispose d'une fonction permettant à un programme (le parent) de lancer l'exécution d'un autre programme (le fils).

La fonction **4Bh** de l'interruption **21h** exécute un programme fils.

Il est aussi possible d'envoyer des informations au programme fils, et ce dernier pourra, à son tour, renvoyer au parent un code indiquant "si tout s'est bien passé".

- La fonction **4Ch** (de l'interruption 21h) permet à un programme fils de se terminer en envoyant au parent un code.

- La fonction **4Dh** (de l'interruption 21h) permet au parent de récupérer un code venant d'un programme fils.

Exemple de programmation

L'utilisation de cette fonction **4Bh** est un petit peu plus compliquée qu'un appel simple. En effet, cette fonction détruit **tous** les registres, excepté CS et IP. Nous ne pouvons donc pas utiliser les fonctions internes de TURBO PASCAL ("intr" et "Msdos") car elles font référence aux registres de segment; ce qui n'est pas une bonne idée si ces registres sont détruits! Pire, comment vont-elles pouvoir revenir au programme puisque l'adresse de retour est sur un stack inaccessible (plus de SS, SP, BP) ? Nous allons donc utiliser le code INLINE. De plus, il n'y a **qu'un seul** endroit où nous pouvons sauver les registres : le code segment. En effet, CS est le seul registre (de segment) qui n'est pas détruit. On utilisera donc des "typed constants" (constantes avec type); ce sont des constantes qui sont traitées comme des variables et qui sont stockées dans le CODE SEGMENT (CS) :

```
 Function F4B(NomduFils : Str255) : Integer;
{Cette fonction exécute le programme "nomdufils"}

{définition des "typed constants"}
 Const
   SauveSS : Integer = 0;
   SauveSP : Integer = 0;
   SauveBP : Integer = 0;
   SauveDS : integer = 0;

 Var
   Regs : Registres;
   FCB1, FCB2 : array[0..36] of Byte;
   Chemin : Str255;
   Parametres : Str255;
   ParmTable : record  {Table de paramètres}
                 AdrEnv : Integer;
```

```
                AdrCLin : ^Integer;
                AdrFCB1 : ^Integer;
                AdrFCB2 : ^Integer;
               end;
 rax,rFl: Integer;

begin
 if Pos(' ', NomDuFils) = 0 then
 begin
  Chemin := NomDuFils+#0; {Le chemin doit       }
                          {se terminer  par le  }
                          {caractère ASCI 0     }
  Parametres := ^M;       {Pas de paramètres}
 end
 else
 begin
  Chemin := Copy(NomDuFils, 1, Pred(Pos(' ',
                  NomDuFils)))+#0;
  Parametres := Copy(NomDuFils, Pos(' ',
                      NomDuFils), 255)+^M;
 end;
{ajuste la longueur du string "paramètres"}
 Parametres[0] := Pred(Parametres[0]);

{Crée et initialise 2 FCB}
 with regs do
 begin
  FillChar(FCB1, SizeOf(FCB1), 0);
  Ax := $2901;
  Ds := Seg(Parametres[1]);
  Si := Ofs(Parametres[1]);
  Es := Seg(FCB1);
  Di := Ofs(FCB1);
  MsDos(regs);{ Crée le premier FCB }

  FillChar(FCB2, SizeOf(FCB2), 0);
  Ax := $2901;
  Ds := Seg(Parametres[1]);
  Si := Ofs(Parametres[1]);
  Es := Seg(FCB2);
  Di := Ofs(FCB2);
  MsDos(regs);{Crée le second FCB }

  with ParmTable do
  begin
   AdrEnv := 0;
```

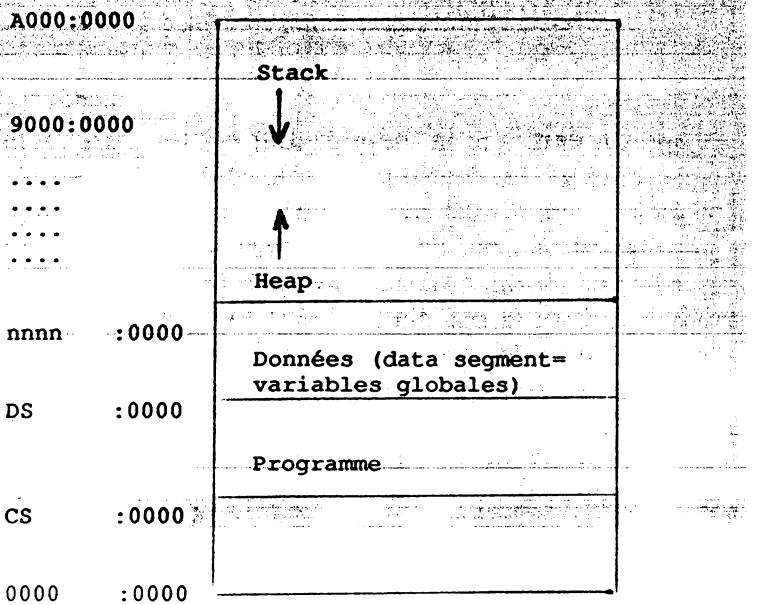
```
    {indique qu'on utilise l'env. du parent}
    AdrCLin := Addr(Parametres);
    AdrFCB1 := Addr(FCB1);
    AdrFCB2 := Addr(FCB2);
  end;

  Inline(
   $8D/$96/>Chemin/       {LEA DX,[BP+>Chemin]     }
   $42/                   {INC DX   ;DS:DX =       }
                          {         ;offset        }
                          {         ;Chemin[1]     }
   $8D/$9E/>PARMTABLE/    {LEA BX,[BP+>PARMTABLE]  }
   $B8/$00/$4B/           {MOV AX,$4B00            }
   $55/                   {PUSH BP  ;sauve BP      }
   $2E/                   {CS:      ;sauve DS,SS   }
                          {         ;et SP         }
   $8C/$1E/>SauveDS/      {MOV [>SauveDS],DS       }
   $2E/                   {CS:                     }
   $8C/$16/>SauveSS/      {MOV [>SauveSS],SS       }
   $2E/                   {CS:                     }
   $89/$26/>SauveSP/      {MOV [>SauveSP],SP       }
   $16/                   {PUSH SS  ;assigne SS à  }
   $1F/                   {POP DS   ;...DS         }
   $16/                   {PUSH SS  ;assigne SS à  }
   $07/                   {POP ES   ;...ES         }
   $CD/$21/               {INT $21  ;Appelle DOS   }
   $FA/                   {CLI      ;empêche ints  }
   $2E/                   {CS:      ;réinit SP,SS  }
                          {         ;et DS         }
   $8B/$26/>SauveSP/      {MOV SP,[>SauveSP]       }
   $2E/                   {CS:                     }
   $8E/$16/>SauveSS/      {MOV SS,[>SauveSS]       }
   $FB/                   {STI      ;autorise int  }
   $2E/                   {CS:                     }
   $8E/$1E/>SauveDS/      {MOV DS,[>SauveDS]       }
   $5D/                   {POP BP   ;récupère BP   }
   $9C/                   {PUSHF    ;assigne FLAG  }
                          {         ;à...          }
   $8F/$86/>rFL/          {POP [BP+>rFL] ;...fl    }
   $89/$86/>rAX);         {MOV [BP+>rAX],AX        }
  if (RFl And 1)<>0 then {teste le bit CARRY}
   F4B := Rax {code d'erreur dans ax}
  else
   F4B := 0;
 end;{with regs}
end;{Function F4B}
```

7. Une application

Quand on explique la théorie, tout est toujours facile mais parfois la pratique révèle d'importants problèmes. Nous allons maintenant découvrir comment un programme écrit en Turbo Pascal peut en exécuter un autre et comment faire pour qu'il puisse accéder au DOS. L'exemple est particulièrement intéressant car l'organisation de la mémoire de Turbo Pascal est un peu spéciale.

Les deux fonctions du MS-DOS qui seront utilisées sont la fonction **4Ah** pour réduire la mémoire possédée avant l'appel et la récupérer après, et la fonction **4Bh** pour l'appel au programme fils.

Si l'on se réfère aux fonctions nécessaires, tout a l'air fort simple. Malheureusement, il y a un hic... Un programme Turbo Pascal gère un *stack* pour l'appel aux sous-programmes ainsi qu'une autre structure appelée HEAP pour les variables dynamiques (créées par New, GetMem, Mark). Le *heap* possède toutes les caractéristiques d'une pile mais, contrairement au stack (qui empile les éléments par le bas), le heap les empile par le haut. Lorsqu'un programme est chargé en mémoire, Turbo Pascal installe le stack (le hic) tout en haut de la mémoire (il peut le faire puisqu'il possède toute la mémoire) et le HEAP juste au-dessus du programme. Ce qui donne le schéma suivant — les flèches représentent la direction par où les piles "grandissent" :



Il est donc possible qu'à un certain moment les deux piles se touchent (cas où il y a énormément de variables dynamiques), Turbo Pascal enverra alors un message "STACKS COLLISION" (run-time error FF) et arrêtera le programme. Cette situation de collision est peu fréquente et il y aura en général beaucoup de mémoire libre entre les deux structures. Malheureusement, ce n'est pas très utile puisqu'il est impossible d'exécuter un programme fils en l'insérant entre les deux piles; il faudrait donc se "débarrasser" du stack (en haut de la mémoire) et libérer toute la mémoire au-dessus du sommet du *heap*. Comme les données du stack doivent bien être sauvegardées "quelque part" et qu'un *stack* n'est en général pas très important en taille, on le sauvegardera dans le DATA SEGMENT

du programme parent. On installera ensuite un nouveau stack juste au dessus du *heap*. Voici le détail des opérations à réaliser :

1. Pour calculer le nombre de bytes à sauver, la première instruction du programme (parent) devra lire la valeur du registre SP (qui pointe juste avant le début du stack puisqu'il n'y a rien dedans au démarrage) et devra sauver cette valeur dans une variable globale (ces variables sont dans le DS d'un programme Pascal).

2. La première instruction de la procédure qui se charge de donner le contrôle au programme fils ("AppelleFils" dans notre exemple) devra relire la valeur de SP et la sauvegarder. La différence entre ce SP et celui de début du programme donne le nombre de bytes actuellement utilisés par le stack.

3. Il faut aussi sauvegarder SS dans une variable globale. Cette valeur et le SP (du point 2) seront utilisés pour remettre, à la fin de l'opération, le stack en haut de la mémoire.

4. On peut lire le nombre de paragraphes que l'on possède grâce au MCB qui se trouve juste avant le programme en mémoire (cf. le chapitre sur l'organisation de la mémoire).

5. On calcule ensuite le nombre de paragraphes (un paragraphe = 16 bytes) que l'on doit conserver :

```
Adresse du segment situé juste au-dessus du
```

```
    sommet du HEAP.
-
    Cseg
+
    Nombre de bytes du nouveau stack (ce nombre
    est déterminé par nous).
```

Une variable interne de Turbo Pascal — HEAPPTR — nous indique l'adresse de segment du sommet du *heap*.

6. Si l'on soustrait la valeur calculée au point 5 de celle calculée au point 4, on obtient (4-5); ce qui représente le nombre de paragraphes que l'on peut libérer pour le programme fils.

7. Il faut tester que la taille du stack ne dépasse pas la taille de la zone prévue pour le sauver (dans DS).

8. Il faut tester que le nombre de paragraphes que nous pouvons libérer est suffisant pour l'opération.

9. Si l'un des tests 8 ou 9 signale un manque de mémoire, l'opération se termine; sinon on continue.

10. Sauvegarder l'image de l'écran dans une variable **globale**, l'effacer et éventuellement afficher un message.

11. On peut transférer les données du stack dans la zone prévue du DS (un array GLOBAL).

12. Il faudra ensuite recréer le nouveau stack (on en a encore besoin pour l'appel à certaines procédures...) juste au-dessus du *heap* (qui ne grandira plus puisqu'on ne crée pas de variables dynamiques pour le moment).

13. Appel de notre fonction "ajustetaille" pour réduire la mémoire allouée.

14. Appel de notre fonction "F4b" pour exécuter le programme Fils.

15. Rétablir SS et SP à leurs valeurs originales et recopier notre buffer dans le stack (on remet le stack en haut de la mémoire).

16. Récupérer l'image de l'écran.

17. C'est fini !

Au moment où le programme fils reçoit le contrôle, le schéma de la mémoire est le suivant :

```
A000:0000


9000:0000 Mémoire disponible pour le fils.

....
....
....        Nouveau Stack.
....
            Heap.
nnnn:0000
            Données (c'est ici qu'il y a
                     la copie du stack).
```

```
DS  :0000
          Programme.
CS  :0000

0000:0000
```

Voici maintenant le programme complet :

```
PROGRAM xyz;
Const
 ScreenAdr:Integer=$B000;  {mettre $B800 si    }
                           {écran graphique    }
 TailleStackBuffer = 512;  {Taille de la       }
                           {zone où on         }
                           {sauve le stack     }
 TNStack=1700;       {Taille du nouveau stack  }
 ParasPourDos=1300; {nbr de para. min. pour DOS}

Type
 Str255=String[255];

Var
 DSP,ASS,ASP:Integer;  {SP du début,           }
                       {Ancien SS,             }
                       {Ancien SP              }
 NSS,NSP:Integer;      {NouveauSS,NouveauSP    }

 NA,NG,NL:integer;     {Paragraphes que nous   }
                       {avons, que nous gardons,}
                       {que nous libérons      }

 ExecPgm:Str255;
 {zone de sauvegarde du stack:}
 StackBuffer:array[1..TailleStackBuffer]
                    of Byte;
 StAvecEnv : Str255;
 Ecran:array[1..4000] of byte;
 ExecStatus: Byte;
 Ch: Char;

procedure SauveEcran;
begin
 Move(Mem[Screenadr:0], Ecran, SizeOf(Ecran));
end;
```

```
procedure Restore Ecran
begin
  Move(Ecran, Mem[Screenadr:0], SizeOf(Ecran));
end;

Procedure AppelleFils(st:str255);

begin
   {Sauve SS dans SauveSS et SP dans SauveSP}
 inline(
   $8C/$D0/      {MOV AX,SS}
   $A3/ASS/      {MOV ASS,AX}
   $89/$26/ASP   {MOV ASP,SP}
 );

  {nouveau stack va juste au dessus   }
  {(succ) du heap                     }
 NSS := Succ(Seg(HeapPtr^));
 NSP := TNStack; {premier él. d'un stack est }
                 {à l'adr. mémoire la plus haute}

  {le nombre de paras que nous avons (NA)     }
  {est dans le MCB qui précède le programme   }
 NA := MemW[Pred(CSeg):3];

   {Nous gardons la différence entre NSS      }
   {et Cseg + Taille du nouveau stack,        }
   {shr 4 pour avoir en paragraphes(/16)      }
 NG := Succ(NSS-CSeg)+Succ(TNStack shr 4);
 NL := NA-NG;{nombre de paras pour le fils}

  {Regarde si le nombre de paras que          }
  {nous pouvons libérer est suffisant....     }
 if (NL> 0) and (NL< (ParasPourDos)) then
 begin
  writeln('Pas assez de mémoire pour accéder
           au DOS');
  exit;
 end;
```

```
    {Regarde si notre buffer est assez          }
    {grand pour sauver le stack...              }
  if succ(DSP-ASP) > TailleStackBuffer then
  begin
    writeln('Pas assez de mémoire pour
      sauver le contexte de ce programme');
    exit;
  end;
ExecPgm:=St;

  (* *****, cf remarque page 279 *)

  {Sauve l'image de l'écran et l'efface}
 SauveEcran;
 ClrScr;
 WriteLn('Taper EXIT pour revenir
          au programme...');

  {Copie le stack dans notre buffer}
 Move(Mem[ASS:ASP], StackBuffer,
      succ(DSP-ASP));

  {installe le nouveau stack: init SS et SP}
 inline(
  $FA/            {CLI }
  $A1/NSS/        {MOV AX,NSS}
  $8E/$D0/        {MOV SS,AX}
  $8B/$26/NSP/    {MOV SP,NSP}
  $FB             {STI }
 );

  {Libère la mémoire, ne garde que ce         }
  {qui est nécéssaire                         }
 AjusteTaille(NG);

  {exécute le prog fils}
 ExecStatus := F4b(St);

  {Récupère toute la mémoire que l'on         }
  {possédait                                  }
 AjusteTaille(NA);

  {Récupère SS et SP}
 inline(
  $FA/            {CLI }
  $A1/ASS/        {MOV AX,ASS}
```

```
    $8E/$D0/        {MOV SS,AX}
    $8B/$26/ASP/    {MOV SP,ASP}
    $FB             {STI }
  );

    {Recopie notre buffer dans le stack}
  Move(StackBuffer, Mem[ASS:ASP],
       succ(DSP-ASP));

    {Pause si on veut que l'utilisateur        }
    {lise un éventuel message de fin du        }
    {programme fils                            }
  Write('taper sur une touche pour
         continuer ');
  Read(Kbd, Ch);

    {recupère l'image de l'écran}
  RestoreEcran;

  if ExecStatus <> 0 then
   Case ExecStatus OF
     $02:writeln('Fichier pas trouvé');
     $03:writeln('Chemin pas trouvé');
     $08:writeln('Mémoire insuffisante');
     $0A:writeln('Environnement incorrect');
   else
    writeln('Erreur inconnue');
   end;

end;{appelle fils}

begin {Programme------------------------------}

  {Sauve SP dans DebutSP}
 inline(
  $89/$26/>DSP); {MOV [>DSP],SP}

 {
     - insérer ici n'importe quel programme -
 }

  {exemple d'appel (cet appel peut aussi       }
  {se trouver dans une procédure),             }
  {le programme TURBO.COM sera exécuté:        }
 AppelleFils('TURBO.COM');
```

```
writeln;
write('Je suis revenu de TURBO.COM');

 {il peut aussi y avoir un chemin :}

AppelleFils('C:\FRAMEII\FW.EXE');

end.
```

Tel qu'il est, ce programme est très bien mais il y a moyen de faire mieux. Pourquoi ? Parce que dans la version actuelle, si on ne spécifie pas de chemin, le système ne cherchera le programme que dans le répertoire courant, il n'utilisera donc pas le PATH. De plus, il n'y a pas moyen d'exécuter des fichiers .BAT.

Heureusement, il y a une solution très simple: nous allons demander à COMMAND.COM de faire cela pour nous. Cependant, il faut savoir dans quel répertoire le programme se trouve. Cette information fait partie de l'environnement (voir le chapitre consacré à l'environnement); il faudra rechercher la chaîne de caractères ("string") "COMSPEC=", laquelle est suivie du chemin et du nom du processeur de commandes (qui peut être autre que COMMAND.COM). Ensuite, il faudra construire une chaîne de caractères qui a le format suivant :

```
xxx /C NomDuFils
```

où *xxx* est ce qui se trouve après COMSPEC dans l'environnement (exemple : C:\COMMAND.COM) et *NomduFils* est le nom, **sans** extension, du programme à exécuter.

Exemples:

```
C:\COMMAND.COM /C TURBO
C:\COMMAND.COM /C FW
```

C'est cette chaîne de caractères qui sera transmise à notre fonction F4B.

Remarque: Dans le cas où on souhaite exécuter un accès DOS, le programme à exécuter est justement COMMAND.COM, il ne faudra donc pas encore ajouter /C COMMAND. On peut, par exemple, convenir que si le nom du programme transmis à "AppelleFils" est "(rien)", il faut appeler COMMAND.COM.

En supposant que nous ayons une fonction "ChercheDansEnv" qui recherche "COMSPEC" dans l'environnement et qui renvoie le nom (C:\COMMAND.COM), on ajoutera donc le code suivant dans notre programme à l'endroit marqué "*****" — voir page 276 — (cf. le chapitre consacré à l'environnement : on y trouve un exemple de consultation de l'environnement).

```
StAvecEnv:= ChercheDansEnv('COMSPEC');
if St <> '' then
 StAvecEnv := StAvecEnv+' /C '+St;
```

L'appel à 4FB devient :

```
 {exécute le prog fils}
ExecStatus := F4b(StAvecEnv);
```

Et l'appel à AppelleFils :

```
AppelleFils(FW);
AppelleFils(Turbo);
AppelleFils(''); {pour un accès DOS}
```

Note: Le concept de variable **globale** ou **locale** est extrêmement important. Les variables locales sont sauvées sur le stack tandis que les variables globales sont sauvées dans le DATA segment.

8. Le CTRL-C et le CTRL-Break

Il est assez aberrant de constater que beaucoup de programmes sur le marché peuvent encore être interrompus par un simple CTRL-C ou CTRL-BREAK et cela quelle que soit la situation, critique ou non, dans laquelle ils se trouvent. Il faut savoir qu'en cas de CTRL-C, le microprocesseur interrompt le programme en cours pour donner le contrôle à l'interruption **23h** (interruption **1Bh** dans le cas de CTRL-BREAK) qui aura normalement pour effet de retouner au DOS. L'interruption d'un programme peut avoir des conséquences graves :

- Si le programme interrompu avait modifié une interruption, il est évident que tout appel ultérieur à cette interruption provoquera le branchement à une adresse à laquelle le programme ne se trouve peut-être plus puisqu'il a été interrompu.

- Les fichiers ouverts par le programme interrompu ne sont pas fermés par un CTRL-C, il y a donc un important risque de perte d'informations (cf. le chapitre 5 : Manipulation de fichiers).

Tout programme correct devrait donc réécrire ces deux interruptions pour éviter un arrêt du programme non contrôlé. La méthode la plus simple pour empêcher l'utilisateur de se servir de ces touches est de créer deux interruptions qui ne font rien. Ainsi, le CTRL-C et le CTRL-BREAK seront purement et simplement ignorés par le programme.

Note : L'interruption 23h (CTRL-C) est appelée par le DOS tandis que l'interruption 1Bh (CTRL-BREAK) est appelée par le BIOS.

Exemple de programmation

En Turbo Pascal, on pourra aussi éviter qu'un "runtime error" n'interrompe le programme. Tous les cas d'erreur critique sont traités par la même procédure qu'il est possible de réécrire.

```
Procedure ErreursTurbo(Num,adr:Integer);


begin
 write(^G^G);
 Writeln;
 Writeln('Erreur :',Lo(Num));
 Writeln;
 Case Hi(Num) of
  0:Writeln('Ctrl-c ou Ctrl-Break');
  1:Writeln('Erreur de fichier (I/O)');
  2:Writeln('Erreur de programme (Runtime)');
 end;

 {insérer ici toutes les opérations à effectuer  }
 {avant de terminer (fermer les fichiers,...)    }

 halt;
end;{ErreursTurbo}
```

Et on ajoutera la ligne suivante parmi les premières du programme :

```
ErrorPtr:=Ofs(ErreursTurbo);
```

Annexe 1

Les codes d'erreur du MS-DOS

Après chaque opération sur les fichiers et les périphériques, on pourra appeler la fonction 59h de l'interruption 21h qui permet d'obtenir une information plus précise. Cette fonction n'est disponible que pour les versions MS-DOS 3.0 et supérieures.

Code de l'erreur (dans AX) :

00h	Pas d'erreur.
01h	Numéro de fonction invalide.
02h	Fichier non trouvé.
03h	Chemin non trouvé.
04h	Trop de fichiers ouverts, plus de handle disponible.
05h	Accès refusé.
06h	Handle incorrect.
07h	MCB détruit.
08h	Mémoire insuffisante.
09h	Adresse de zone incorrecte.
0Ah	Environnement incorrect.
0Bh	Format incorrect.
0Ch	Code d'accès incorrect.
0Dh	Donnée incorrecte.
0Eh	Réservé.
0Fh	Référence de disque incorrecte.
10h	Tentative de destruction du répertoire courant.
11h	Périphériques différents.
12h	Plus de fichiers.
13h	Disque protégé contre l'écriture.
14h	Unité inconnue.

15h	Disque pas prêt.
16h	Commande incorrecte.
17h	Erreur de mémoire.
18h	Longueur de structure incorrecte.
19h	Erreur de positionnement du pointeur de fichier.
1Ah	Disque non MS-DOS.
1Bh	Secteur non trouvé.
1Ch	Plus de papier dans l'imprimante.
1Dh	Erreur d'écriture.
1Eh	Erreur de lecture.
1Fh	Erreur générale.
20h	Violation de réseau (partage).
21h	Violation de bloquage (lock).
22h	Changement de disque incorrect.
23h	FCB non disponible.
24h	Tampon de partage saturé.
25h à 31h	Réservés.
32h	Requête de réseau non supportée.
33h	Machine à distance n'est pas prête.
34h	Nom déjà existant sur le réseau.
35h	Nom du réseau non trouvé.
36h	Réseau occupé.
37h	Périphérique n'existe plus sur le réseau.
38h	Nombre de commandes du BIOS réseau saturé.
39h	Erreur matériel dans la connexion du réseau.
3Ah	Réponse de réseau incorrecte.
3Bh	Erreur de réseau.
3Ch	Adaptateur incompatible.
3Dh	Queue d'impression pleine.

3Eh	**Queue d'impression non remplie.**
3Fh	**Plus de place pour le fichier d'impression.**
40h	**Nom du réseau effacé.**
41h	**Accès refusé.**
42h	**Type de périphérique incorrect.**
43h	**Nom de réseau non trouvé.**
44h	**Limite du nom du réseau saturée.**
45h	**Session du NETBIOS saturée.**
46h	**Pause temporaire.**
47h	**Requête non acceptée.**
48h	**Redirection de l'imprimante ou du disque temporairement suspendue.**
49h à 4Fh	Réservés.
50h	Fichier existant déjà.
51h	Réservé.
52h	Pas possible de créer un répertoire.
53h	Erreur d'interruption 24h.
54h	En dehors de la structure.
55h	Déjà assigné.
56h	Mot de passe incorrect.
57h	Paramètres incorrects.
58h	Erreur d'écriture.

Classe de l'erreur (dans BH) :

01h	Plus de ressource (mémoire...).
02h	Situation temporaire, devrait se terminer rapidement.
03h	Problème d'autorisation.
04h	Erreur interne dans le programme système.
05h	Erreur matérielle.
06h	Erreur système. Fichier de configuration incorrect ou inexistant (Le programme en cours n'est pas en faute).
07h	Erreur du programme en cours.
08h	Fichier non trouvé.
09h	Fichier avec type ou format incorrect ou inapplicable.
0Ah	Fichier interbloqué.
0Bh	Problème avec la disquette.
0Ch	Déjà existant.
0Dh	Inconnu.

Action suggérée (dans BL) :

01h	Réeffectuer quelques essais avant de demander à l'utilisateur de corriger l'erreur.
02h	Réeffectuer quelques essais espacés dans le temps avant de demander à l'utilisateur de corriger l'erreur.
03h	Demander à l'utilisateur d'entrer une information correcte (lettre d'un disque, nom d'un fichier).

04h Effectuer les opérations de fin et terminer le programme.
05h Terminer le programme immédiatement.
06h Ignore l'erreur.
07h Demander à l'utilisateur de corriger la cause de l'erreur (changer le disque) et réessayer.

Localisation de l'erreur (dans CH) :

01h Inconnu.
02h Périphérique bloc.
03h Réseau.
04h Port série.
05h Mémoire.

Les fonctions suivantes font partie de l'interruption 21h (Appel à MS-DOS).

1. Fonctions clavier

a. Fonctions d'entrée de caractères

01h - Entrée de caractères avec affichage à l'écran.
03h - Entrée auxiliaire.
06h - Entrée directe à la console.
07h - Entrée de caractères sans filtre avec affichage.
08h - Entrée de caractères sans affichage.
0Ah - Entrée de caractères avec temporisation.
0Bh - Test de l'état du clavier.
0Ch - Vide le buffer et lecture du clavier.

b. Fonctions de sortie de caractères

02h - Sortie de caractères.
04h - Sortie auxiliaire.
05h - Impression d'un caractère.
06h - Affichage direct à la console.
09h - Affichage d'une chaîne de caractères.

2. Fonctions de gestion des disques

a. Fonctions disque

0Dh - Reset du disque.
0Eh - Selection d'un disque.
19h - Lecture du disque courant.
1Bh - Lecture des paramètres du disque courant.
1Ch - Lecture des paramètres d'un disque.
2Eh - Active, Désactive le drapeau de vérification.
36h - Lecture de l'espace disque disponible.
54h - Lecture de l'état du drapeau de vérification.

b. Fonctions de gestion des fichiers

0Fh - Ouverture d'un fichier avec FCB.
10h - Fermeture d'un fichier avec FCB.
11h - Trouve le premier fichier d'un répertoire.
12h - Trouve le fichier suivant.
13h - Effacement d'un fichier.
16h - Création d'un fichier avec FCB.
17h - Renomme un fichier.
1Ah - Spécifie l'adresse de la DTA.
23h - Lecture de la taille d'un fichier.
2Fh - Lecture de l'adresse de la DTA.
3Ch - Création d'un fichier avec "handle".
3Dh - Ouverture d'un fichier avec "handle".
3Eh - Fermeture d'un fichier.
41h - Effacement d'un fichier.

43h - Lecture/écriture des attributs d'un fichier.
45h - Duplique un "handle".
46h - Force la duplication d'un "handle".
4Eh - Trouve le premier fichier dans un répertoire.
4Fh - Trouve le fichier suivant.
56h - Renomme un fichier.
57h - Lecture/écriture de la date et l'heure d'un fichier.
5Ah - Création d'un fichier temporaire.
5Bh - Création d'un nouveau fichier.
5Ch - Lock/unlock une partie d'un fichier.

c. Fonctions de gestion des données

14h - Lecture séquentielle.
15h - Ecriture séquentielle.
21h - Lecture aléatoire.
22h - Ecriture aléatoire.
24h - Sélection d'un enregistrement relatif.
27h - Lecture aléatoire d'un bloc.
28h - Ecriture aléatoire d'un bloc.
3Fh - Lecture d'un fichier ou d'un périphérique.
40h - Ecriture sur un fichier ou sur un périphérique.
42h - Positionnement du pointeur de fichier.

d. Fonctions de gestion des répertoires

39h - Création d'un nouveau répertoire.
3Ah - Suppression d'un répertoire.

3Bh - Change le répertoire courant.
47h - Lecture du répertoire courant.

3. Fonctions de gestion des programmes

00h - Terminaison d'un programme.
31h - Terminaison mais reste résidant.
4Bh - Charge et exécute un programme (EXEC).
4Ch - Terminaison avec code de retour.
4Dh - Lecture des codes de retour envoyés par le programme fils.
59h - Lecture des codes d'erreur étendus.

4. Fonctions de gestion de la mémoire

48h - Allocation d'un bloc de mémoire.
49h - Libération d'un bloc de mémoire.
4Ah - Réajuste la taille d'un bloc de mémoire.
58h - Lecture/sélection de la stratégie d'allocation mémoire.

5. Fonctions générales

25h - Ecriture d'un vecteur d'interruption.
26h - Création d'un nouveau PSP.
29h - Analyse d'un nom de fichier.
2Ah - Lecture de la date.
2Bh - Ecriture de la date.
2Ch - Lecture de l'heure.
2Dh - Ecriture de l'heure.
30h - Lecture de la version du MS-DOS.
33h - Lecture/écriture du drapeau de CTRL-C.
34h - Lecture de l'adresse du drapeau InDos (compteur de l'interruption 21h).
35h - Lecture du vecteur d'interruption.
38h - Lecture/écriture du pays courant.
44h - IOTCL.
5Eh - Nom de la machine en réseau, installation de l'imprimante.
62h - Lecture de l'adresse du PSP.

ANNEXE 3

GLOSSAIRE

BIOS
Partie du système d'exploitation contenant des sous-programmes de gestion des périphériques directement dépendante du type de matériel installé.

Cluster
Un cluster est l'unité d'allocation d'espace disque à un fichier, il est composé de plusieurs secteurs.

DCB
Abréviation de "Device Control Block". Un DCB est une structure que MS-DOS utilise pour la gestion des fichiers (méthode des "handles").

DTA
Zone de mémoire utilisée par MS-DOS lors de certaines opérations sur disque.

Enregistrement
Un fichier contient un ensemble d'informations d'un même type. Il sera alors divisé logiquement en enregistrements, chaque enregistrement contiendra une des informations.

Environnement
Zone mémoire utilisée pour le dialogue entre programmes.

FAT
La FAT (File Allocation Table) est une table qui contient, pour chaque fichier, la liste des clusters contenant les informations de celui-ci.

Fermeture d'un fichier
Fermer un fichier est une action qui signale au système d'exploitation que le programme a terminé son travail sur le fichier.

Fichier
Structure sur disque contenant un ensemble d'informations.

Fils
On appelle programme "fils" (child process) un programme qui est exécuté par un autre. Tous les programmes exécutés sur les IBM PC sont des fils de COMMAND.COM.

Fonction
Une interruption pourra proposer plusieurs services, chacun d'eux est appelé "fonction".

Handle
Un handle est un numéro qui est associé à un fichier. Ce numéro sera attribué par MS-DOS lors de l'ouverture du fichier et il devra être spécifié lors de toutes les opérations sur celui-ci.

Interruption.
Une interruption est un sous-programme qui sera exécuté sur demande du microprocesseur, d'un périphérique ou d'un programme. Les services du BIOS et du DOS sont des interruptions.

MCB
Abréviation de "Memory Control Block". Un MCB est une structure de 16 octets utilisée par MS-DOS pour la gestion des blocs mémoire.

MS-DOS
Partie du système d'exploitation servant d'intermédiaire entre les applications et les routines du BIOS. Le MS-DOS s'occupe aussi de l'organisation des fichiers sur disque, de la mémoire, etc.

Ouverture d'un fichier
Ouvrir un fichier est une action qui signale au système d'exploitation que le programme souhaite consulter ou modifier les informations contenues dans ce fichier.

Paragraphe
Un "paragraphe" est un ensemble de 16 octets de mémoire. On parlera aussi d'adresse d'une zone sur une "limite de paragraphe", cela signifie que l'adresse du premier octet de la zone doit être divisible par 16.

Parent
On appelle "programme parent" (Parent process) un programme qui en exécute un autre.

Périphérique
Elément d'un ordinateur lui permettant de communiquer avec le monde extérieur. On distingue les périphériques d'entrée (clavier, souris,...) et les périphériques de sortie (écran, imprimante,...).

Pile
Zone de la mémoire utilisée pour la sauvegarde de valeurs et pour la transmission de paramètres aux sous-programmes.

PSP
Zone de 256 octets située avant chaque programme en mémoire. Elle contient des informations utiles au déroulement du programme.

Registre
Emplacement de mémoire à l'intérieur du microprocesseur.

Réseau local
Il est possible de connecter plusieurs ordinateurs IBM PC entre eux pour qu'ils puissent échanger des informations ou partager des ressources (disque, imprimante; etc.), on parle alors de réseau local.

Résident (programme)
Après qu'il soit installé en mémoire, ce type de programme rend le contrôle de l'ordinateur à l'utilisateur qui pourra exécuter un autre programme. On distingue deux catégories de programmes *résidents* : les programmes "passifs" qui sont activés sur demande du programme en cours et les programmes "actifs" qui ne sont pas liés au programme en cours (ils redirigent une interruption).

Secteur
Un secteur est l'unité de base d'accès à un disque, il contient 512 octets.

Stack
Voir "Pile".

Système d'exploitation
Ensemble de programmes chargés de gérer au mieux l'ensemble des ressources d'un ordinateur. Il regroupe le BIOS, le DOS et l'interface utilisateur.

Vecteur d'interruption
Emplacement mémoire contenant l'adresse d'une routine d'interruption. L'ensemble des vecteurs est stocké au début de la mémoire dans une zone appelée "table des vecteurs d'interruption".

Index

8259 23
Attributs de fichiers 114,220
BIOS 13, 14
Blocs (périphériques) 78
Caractères (périphériques) 52
CGA 54,90
Chemin 109,278
Clavier 64,72,139
Cluster 203,210,220,233,238
Contrôleur d'interruption 23
Critique (erreur) 96,282
Curseur 56,90
Cylindre 189,191
DCB 174
Disque 79,97,130,188,192
DOS 13, 15
DTA 111,120,121,257
Ecran 54,72,139
Effacé (fichier) 220,233
Enregistrement 138,147
Environnement 250,256,259,278
Face 188
FAT 203,210,219,233,236
FCB 40,256
Fichier 137,219,233
Fils (programme) 244,255,259,265,271
Filtre 74
Fonction 21
Handle 41,72,139,174,256,258
Heap 269

Imprimante 69,72,97,139
INT 28
INTR 28,196,266
Interruption 21
IRET 33
Logicielles (interruptions) 25
Logique (fichier) 137
Logique (numérotation) 190,197,236
Masque 110,114,130,216,230
Matérielles (interruptions) 23
MCB 245,247,271
Memory allocation error 246,247
Modification d'une interruption 31
Niveau d'interruption 24
Paragraphe 247,256,271
Parent (programme) 244,255,259,265,271
Parquer (les têtes d'un disque dur) 80
Partition 197,199
Périphérique 23,51,72,139
Physique (fichier) 137
PIC 23
Pile Cf. Stack
Piste 188
Pointeur de fichier 147
Priorité 24
PSP 111,121,174,247,250,255,259
Retour d'interruption 33
Réelle (numérotation) 190,197
Répertoire 108,132,137,150,211,219,223,237
Réseau local 42,141,170

Résidente (routine) 34,250
RS232 60
Scan Code 64,67
Secteur 79,97,188,190,194,197,203,220
Série (port) 58,72,139
Souris 82
Stack 30,196,269
Système d'exploitation 13
Taille d'un fichier 148,150,160
Vecteur d'interruption 27,245
Vidéo 54

Table des matières

Sommaire 5
Préface 7

Chapitre 1 : Le système d'exploitation 11
1. Introduction 13
2. Le BIOS 14
3. Le MS-DOS 15

Chapitre 2 : Les interruptions 19
Les interruptions "processeur" 22
Les interruptions périphériques 23
Les interruptions "programmes" 25
Les vecteurs d'interruptions 27
Appel d'une interruption 28
Modification d'une interruption 31
Exemple de programmation 34
Evolution de MS-DOS 39
Organisation de la mémoire 44
Récapitulation 46

Chapitre 3 : Les périphériques 49
1. Introduction 51
2. Les périphériques "caractères" 52
L'interruption 10h 54
Exemples de programmation 56
L'interruption 14h 58
Exemples de programmation 60
L'interruption 16h 64
Exemples de programmation 65
L'interruption 17h 69
Exemples de programmation 70
Les fonctions 3Fh et 40h de l'int. 21h 72
Exemples de programmation 74

3. Les périphériques "bloc" 78
L'interruption 13h 79
Exemples de programmation 80
4. Autres périphériques 82
L'interruption 33h 82
Les fonctions de status et de gestion du pointeur de la souris 84
Les fonctions de positionnement de la souris 86
Modification de l'aire de déplacement de la souris 89
Exemples de programmation 90
5. L'interruption 24h 96
Exemples de programmation 98
6. Récapitulation 102

Chapitre 4 : Les répertoires 105
1. Généralités 107
2. Les répertoires 108
3. Les fonctions de recherche de fichiers 110
La DTA 111
L'interruption 21h. fonction 4Eh 114
L'interruption 21h. fonction 4Fh 120
Exemples de programmation 121
4. Les fonctions de gestion des répertoires 132
5. Autres fonctions 133
6. Récapitulation 134

Chapitre 5 : Manipulation des fichiers 135
1. Les fonctions du DOS 137
2. Les enregistrements 138
3. Les "handles" 139
La fonction 59h 144
La fonction 42h 147

La fonction 3Eh 150
La fonction 3Dh 151
La fonction 3Ch 153
La fonction 41h 155
La fonction 3Fh 156
La fonction 40h 158
La fonction 45h 160
Exemples de programmation 161
4. MS-DOS et les réseaux locaux 170
Exemples de programmation 171
5. Limite de 20 fichiers ouverts ? 174
Exemples de programmation 178

Chapitre 6 : Organisation des fichiers sur disque 185
1. Introduction 187
2. Structure du disque 188
3. Analyse du disque 190
4. Les interruptions disque 192
L'interruption 13h 193
Les interruptions 25h et 26h 194
Exemples de programmation 195
5. Découpage d'un disque dur 197
Exemples de programmation 199
6. Organisation du disque 203
La zone de démarrage 204
Exemples de programmation 205
La FAT 210
Les FATS à 12 et 16 bits 212
Exemples de programmation 213
Consultation d'une FAT 215
Exemples de programmation 218
Les répertoires et les fichiers 219

Exemples de programmation 222
7. Récupération des fichiers effacés 233
Exemples de programmation 236
8. Autres applications 238
9. Récapitulation 240

Chapitre 7 : La gestion de la mémoire et les programmes 241
1. Le processus "parent appelle fils" 243
2. Structure de la mémoire 245
Structure d'un MCB 247
Exemples de programmation 248
3. Stratégies d'allocation 252
4. Structure d'un programme en mémoire 254
Le PSP (Program Segment Prefix) 255
Exemples de programmation 258
L'environnement 259
Exemples de programmation 260
5. La mémoire réservée aux programmes 262
Exemples de programmation 264
6. Exécution d'un programme "fils" 265
Exemples de programmation 266
7. Une application 269
8. Le CTRL-C et le CTRL-Break 281
Exemples de programmation 282

Annexe 1 : Les codes d'erreur du MS-DOS 283

Annexe 2 : Liste par type des fonctions du MS-DOS 291
1. Fonctions clavier 293
2. Fonctions de gestion des disques 294
3. Fonctions de gestion des programmes 296

4. Fonctions de gestion de la mémoire 296
5. Fonctions générales 297

Annexe 3 : Glossaire 299

Index 307

IMPRIMÉ EN FRANCE PAR BRODARD ET TAUPIN
1825G-5 - Usine de La Flèche (Sarthe), le 23-10-1992.

pour le compte des
Nouvelles Editions Marabout
D.L. octobre 1992/0099/315
ISBN 2-501-01214-3

MICRO-
INFORMATIQUE

LA MAITRISE DU

MS-DOS® ET DU BIOS®

Le but du livre

- Expliquer l'organisation interne de l'ordinateur et le fonctionnement des interruptions (ou "services") du MS-DOS.
- Décrire, à l'aide d'exemples pratiques, comment organiser les répertoires, la mémoire, ou encore le pilotage des périphériques.
- Analyser, sous l'angle de la programmation, la gestion des fichiers et leur organisation sur le disque.
- Les exemples de programmation sont rédigés en Assembleur et en Turbo Pascal®.

Le niveau

- Débutants et utilisateurs confirmés.
- Initiation aux langages de programmation.

40 0123 6

9 782501 012140

MARABOUT INFORMATIQUE

M 12

Couverture: Duc